Interface Theories for Causality Analysis in Actor Networks

Ye Zhou

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2007-53
http://www.eecs.berkeley.edu/Pubs/TechRpts/2007/EECS-2007-53.html

May 15, 2007

Interface Theories for Causality Analysis in Actor Networks

by

Ye Zhou

B.E. (Tsinghua University) 2001
M.S. (University of California, Berkeley) 2003

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Engineering-Electrical Engineering and Computer Sciences

in the

GRADUATE DIVISION
of the
UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:
Professor Edward A. Lee, Chair
Professor Alberto Sangiovanni-Vincentelli
Professor Peter J. Bickel

Spring 2007

The dissertation of Ye Zhou is approved:

Chair Date

Date

Date

University of California, Berkeley

Spring 2007

Interface Theories for Causality Analysis in Actor Networks

Ye Zhou

Abstract

Interface Theories for Causality Analysis in Actor Networks

by

Ye Zhou

Doctor of Philosophy in Engineering-Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Edward A. Lee, Chair

This dissertation focuses on 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 dissertation studies formal anal-

ysis of such components to include such causality information. A causality interface framework, which focuses on functional actors, is presented. I show how this causality interface can be algebraically composed so that compositions of components acquire causality interfaces that are inferred from their components and the interconnections. I illustrate the use of these causality interfaces to statically analyze timed models and synchronous language compositions for causality loops and dataflow models for deadlock. I also show that that causality analysis for each communication cycle can be performed independently and in parallel, and it is only necessary to analyze one port for each cycle. Later, I present another framework called ordering dependency. Ordering dependency captures causality properties that are not captured by the functional abstraction of actors. I illustrate the use of ordering dependencies to analyze rendezvous of sequential programs, and its use in scheduling distributed timed systems.

Professor Edward A. Lee
Dissertation Committee Chair

To my parents, Runde Zhou and Shenmei Jin

Contents

List of Figures

Acknowledgments

I am deeply grateful to my advisor, Professor Edward A. Lee. I could have never gone this far to write this dissertation without his constant support. I enjoyed the time when I walked into his office and went through some brainstorms with him. He always showed great interest in my work, and offered insightful comments and encouraging words. He also taught me how to write good papers and be a good presenter.

Professor Alberto Sangiovanni-Vincentelli and Professor Peter Bickel served on my dissertation committee. Professor Bickel, Professor Shankar Sastry and Professor Jan Rabaey served on my qualifying exam committee. Everyone has been very supportive. I would like to thank each of them for the precious time they took from their busy schedules and their constructive suggestions on my research work.

Getting a Ph.D. is never a single-person project. I had the privilege to work with many smart and friendly people. To name only a few, Xiaojun Liu and Stephen Neuendorffer helped me on my early work in the Ptolemy Project. Eleftherios Matsikoudis served as my math consultant and helped me understand set theory better. Yang Zhao was my buddy in the DOP center. Discussion with her eventually inspired my work on ordering dependency. Slobodan Matic and Arindam Chakrabarti showed me different perspectives of interface theories. Discussions with Guang Yang and Qi Zhu were also very helpful. I also had a lot of great time with Adam Cataldo, Elaine Cheong, and many others at Berkeley.

My thanks also go to the CHESS staff, which includes Christopher Brooks, Tracey Richards, Mary Sprinkle and Mary Stewart. Thanks to Ruth Gjerde for being such a helpful graduate assistant. Thanks to Denise Simard for her excellent work as Edward's assistant.

Lastly, I am deeply grateful to my parents. Raising and educating a child like me is no easy job. I thank them for their constant support and devoted love over all these years. Two decades ago, my father dedicated his Ph.D. dissertation to my mother and me. I now dedicate this dissertation to my dear parents.

Chapter 1

Introduction

1.1 Embedded Systems

Embedded systems have never played a greater role in our daily life than today. They reside in aircraft and cars that take us safely to our destinations. They are in our pockets, as cell phones, MP3 players and PDAs. They control our microwave ovens and robot vacuum cleaners, so as to make our homes more comfortable. Embedded systems are changing the way people work, learn, communicate, and think. As the market for embedded systems continue to grow, so does the demand for safer, smarter, and more interactive products. Most of these products involve thousands of software and hardware components. These components interact with each other, as well as with the physical world, sensing temperature and fluid levels, receiving commands from human, controlling mechanical parts of the device, and etc. For example, Figure 1.1

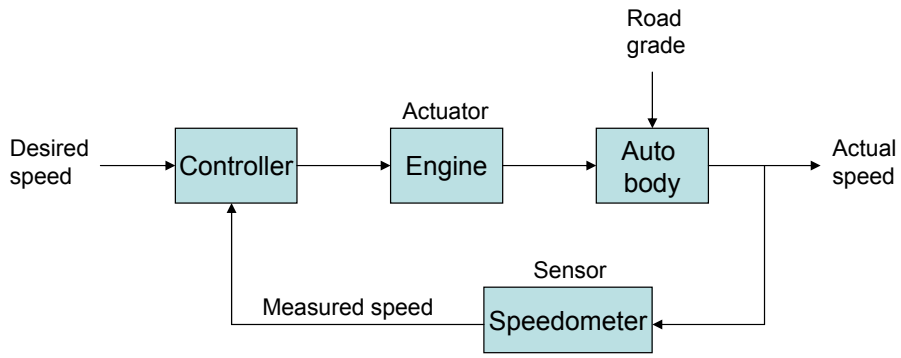


Figure 1.1: Basic block diagram of the cruise control system in automobiles, adapted from [31].

shows the basic block diagram of the cruise control system in automobiles, adapted from [31]. The controller has two inputs, the desired speed set by the driver, and the vehicle's actual speed, which is measured by the speedometer. The controller controls the throttle so as to minimize the difference of the two inputs, despite of interference and disturbance (e.g., road condition).

One of the important issues in embedded software concerns liveness. Programs must not terminate prematurely. For example, the Soft Walls project [20] aims to design embedded systems that continuously inspect and adjust the aircraft's position and velocity so as to prevent the aircraft from flying into no-fly zone. A fire alarm system should not fail to report in occurrence of smoke or fire. Designers must be careful to avoid such pitfalls in system logic, and since such systems usually involve thousands of components, formal design methodologies and analysis are necessary. This dissertation aims to develop theoretic frameworks to reason properties of components, so we can statically determine whether a system is live, whenever it is possible.

1.2 Actor-oriented Design

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* [46]. 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 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" [2, 36] 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 by 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

[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 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.

languages [9], which are used for safety-critical embedded software [13], 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 (semantically) determined by solving fixed point equations. The Lustre [35] and Signal [10] 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 [40], where each component has its own thread of control, extensions to nondeterministic systems [25], and Dennis-style dataflow [26]. In dataflow, components (which are also called "actors" in the original literature) "fire" in response to the availability of input data. The dataflow model of computation has been used in industrial practice, in tools such as SPW from the Alta Group of Cadence (now owned by CoWare), the DSP station from Mentor Graphics, and LabVIEW from National Instruments, as well as in experimental contexts, in frameworks such as Ptolemy developed at Berkeley. Process networks have also been used for embedded system design [25].

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 [48, 45]. Discrete-event (DE) systems [19], for example, are commonly used in circuit design and modeling, such as in Verilog and VHDL [7] languages, and in design of communication networks, such as in OPNET

Modeler[3] and Ns-2[4]. 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 [67], 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 [39, 16, 63], discrete-event formalisms [73, 42], and semantics for continuous-time models [51]. A few formalisms are rich enough to embrace a significant variety of actor-oriented models, including interaction categories [1], behavioral types [50, 6], interaction semantics [65], and the tagged signal model [49].

Some software frameworks also embrace a multiplicity of actor-oriented component architectures, including Reo [5], Ptolemy II [29], PECOS [71], and Metropolis [33]. Finally, a number of researchers have argued strongly for separation between the semantics of functionality (what is computed) and that of interaction between components [18, 41, 34, 69].

[3] http://opnet.com
[4] http://www.isi.edu/nsnam/ns/

1.3 Interface Theories

As in object-oriented design, *composition* and *abstraction* are two central concepts in actor-oriented design. Compositions increase reusability and modularity, and reduce the time spent in design cycles. 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 [5, 60, 22]. In a compositional formalism, a composite actor is itself an actor, and its interface(s) must be those of an actor. A major focus of this dissertation is on how causality properties of composite actors can be determined from their component actors.

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 [72] 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 [47] with inheritance mechanisms.

Following [24] 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 [72]. 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 [50].

In this dissertation, I present an *interface theory* [24], similar in spirit to resource interfaces [21] and behavioral type systems [50], but expressing different properties of systems. The theory captures *causality* properties of actor-oriented designs. Causality properties reflect in the interface the data dependency among ports. The work here is closest in spirit to the component interfaces of Broy in [15], where algebraic compositions of stream functions are formalized. In this dissertation, however, I build 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, the framework I develop 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. By specializing to talk only about causality, we can develop a rich theory that applies across concurrent models of computation.

1.4 Feedback Systems and Introductory Examples

Feedback systems have wide applications in system design since they are robust to varied inputs and disturbance. Feedback systems are used in robust control, such as controllers in automobiles (see Figure 1.1) and computer disk read/write heads

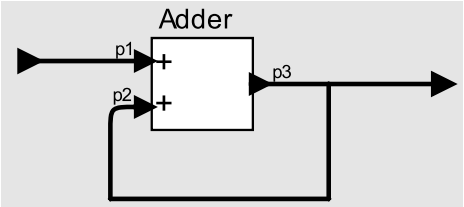


Figure 1.2: A problematic design of an accumulator using an adder actor.

(with accuracy of about a micron), and in tracking, such as surveillance cameras to detect moving objects.

Feedback systems are also used in communication and signal processing, such as estimation and noise cancelation, in adaptive algorithms to allocate power and resources more efficiently [52], and in data compression [57]. More examples can be found in [8, 61, 62].

One of the problems of feedback systems is that their block diagrams have cycles, as we can see from Figure 1.1. This may result in problematic designs. Systems may suffer from causality loops. What properties of an actor determine whether there is a causality loop in the composition? How can we represent such causality properties and compose them, so that looking at the inside details of the composition is unnecessary? I will use some very simple examples to illustrate the problems. It is not hard to imagine that in practical system design, which may involve thousands of components, such problems are more complicated and beyond what human intuition can see. Therefore formal methodology of analysis is necessary.

For example, suppose we want to build an accumulator actor from an adder actor.

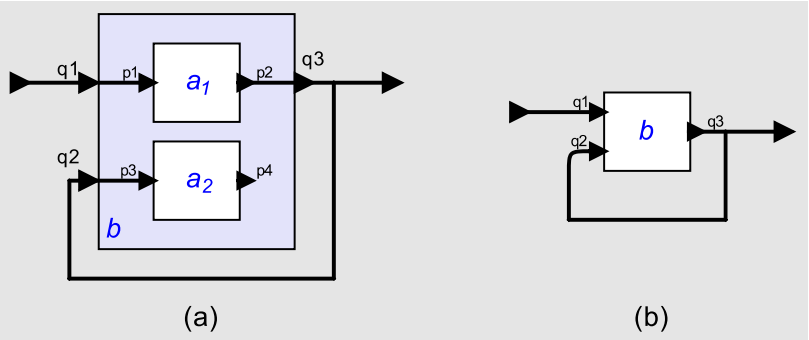


Figure 1.3: A sequential composition of two actors that has the same structural interface of the adder in Figure 1.2. Figure (a) shows the details of the composition, where actors a_1 and a_2 are composed into actor b. Figure (b) hides the details of the composition.

Figure 1.2 shows one possible design, which is problematic. If we view this model as a dataflow model, and the adder takes one token from each input port and produces one token at the output port, then the adder cannot execute since its firing rule is not satisfied. If we view this model as a synchronous/reactive model, the output of the adder remains unknown since the value at port $p2$ at clock tick 0 is unknown. In either case, the feedback loop in the model results in a deadlock, or causality loop. I.e., the output is waiting for the input, while the input is waiting for the output.

We next consider the example shown in Figure 1.3. Two actors a_1 and a_2 are composed into one composite actor b. Figure 1.3(b) shows the view of actor b from the higher level of hierarchy, which has the same structure as in Figure 1.2. Or equivalently, we could say they have the same structural interface, although under the hood, Figure 1.3 is a serial composition. What is the difference between actor b and the adder? Since a composite actor is treated the same as an atomic actor

from the higher level of hierarchy, "*what is under the hood*" must be embodied in the interface of the composite, so that it can differentiate itself from the adder in Figure 1.2. In this dissertation, I will give a formal framework of causality interfaces for actors. This framework captures properties that reflect the difference between Figure 1.2 and Figure 1.3. Causality interfaces are composable, and hence looking into details of the composition is not necessary. The theory also provides a criterion that prevents design pitfalls like the one shown in Figure 1.2. For complicated systems that have more than one feedback loop intersecting with each other, I also prove that under certain models of computation, each feedback loop can be examined independently and in parallel. The theory can be applied to many concurrent semantics such as that of the synchronous languages, discrete-event models, continuous-time models, and dataflow models.

1.5 Overview of the Dissertation

Chapter 2 provides the necessary mathematical background and reviews the tagged signal model [49], a formal framework for considering and comparing actor-oriented models of computation. Actors and their compositions are defined within the tagged signal framework. I also briefly discuss syntaxes for actor models and least fixed point semantics.

Chapter 3 presents the causality interface theory. The first two sections give the basic definition of causality interfaces. A causality interface captures the data depen-

dency of an output port on an input port. How to represent the data dependency among ports and how to use such dependency depends on the semantics (i.e., the model of computation) of the actor network. I present one form of such data dependencies for functional actors. I show that how this interface can be composed and give the liveness condition of an actor network (in the least fixed point semantics). Chapter 4 discusses the application of the causality interface theory to timed systems, including discrete-event models [42, 73], continuous-time models and synchronous languages [64, 12, 27], as well as stream-oriented dataflow models [16, 44].

In Chapter 5, I present another interface framework called *ordering dependency*. This model captures the causality properties that are missing in the functional abstraction of an actor. Therefore, ordering dependency is more general, although it also takes more complicated form. I discuss the use of ordering dependencies to analyze rendezvous of sequential programs, and to schedule distributed timed systems. (Neither case is function-representable.) Conclusions and future work are included in Chapter 6.

Chapter 2

Actors and their Composition

This chapter reviews the *tagged signal model* [49], a formal structure for actors that is sufficiently expressive to embrace all of the models of computation of interest. We will discuss briefly syntaxes that are amenable to actor models and define the visual syntax used in this dissertation. We 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 Mathematical Preliminaries

The tagged signal model and the causality interface theory are built based on theory of partially ordered sets. In this section, I will review some of the mathematical definitions in set theory and establish the definition of *order union* of partially ordered sets. We will be re-visiting these definitions frequently throughout this dissertation.

Definition 2.1 *Let P be a set. A* ***partial order*** *on P is a binary relation $\leq$ on P such that:*

- *(reflexive)* $\forall\ x \in P,$ $x \leq x,$
- *(antisymmetric)* $\forall\ x, y \in P,$ $x \leq y$ *and* $y \leq x \ \Rightarrow\ x = y,$
- *(transitive)* $\forall\ x, y, z \in P,$ $x \leq y$ *and* $y \leq z \ \Rightarrow\ x \leq z.$

A set P with a partial order $\leq$ is called a ***partially ordered set****, or in short, a* ***poset****.*

$\forall\ x, y \in P$, we use $x < y$ to mean $x \leq y$ and $x \neq y$. In contexts that might raise confusion, we use $\leq_P$ to mean the partial order on set P.

A poset $(P, \leq)$ is a *totally ordered set* if $\forall\ x, y \in P$, either $x \leq y$ or $y \leq x$. A totally ordered set is also called a *chain*.

Definition 2.2 *Let $(P, \leq)$ be a poset and Q be a subset of P. We say that $(Q, \leq)$ is a poset* ***induced from*** *$(P, \leq)$ if $\forall\ x, y \in Q$, $x \leq_Q y$ if and only if $x \leq_P y$.*

I.e., Q inherits the partial order relation from P. We say that the partial order $\leq_Q$ is a *projection* of $\leq_P$ onto the subset of Q.

Definition 2.3 *Let S be a subset of a poset P. An element $x \in P$ is a* ***lower bound*** *of S if $\forall\ s \in S$, $x \leq s$. x is the* ***greatest lower bound*** *of S if x is a lower bound of S and for all lower bound y of S, $y \leq x$.*

Upper bound and ***least upper bound*** are defined dually.

We use $\bigwedge S$ to denote the greatest lower bound of S if it exists, and $\bigvee S$ the least upper bound. $x \wedge y$ and $x \vee y$ are alternative notations of $\bigwedge\{x, y\}$ and $\bigvee\{x, y\}$, respectively.

Definition 2.4 *A poset P is a **complete partially ordered set** (CPO) if P has a least element $\perp$, and $\bigvee C$ exists for every chain $C \subseteq P$.*

Definition 2.5 *Let $(P, \leq)$ be a non-empty poset.*

1. *If $\forall\ x, y \in P$, $x \wedge y$ and $x \vee y$ exist, then $(P, \leq)$ is called a **lattice**.*

2. *If $\forall\ S \subseteq P$, $\bigwedge S$ and $\bigvee S$ exist, then $(P, \leq)$ is called a **complete lattice**.*

A complete lattice is therefore a CPO.

2.1.1 Order Union of Posets

In this dissertation, I introduce the definition of *order union* of two posets. The concept of order union will later be used in Chapter 5 to define operations on ordering dependencies.

Let $(P, \leq)$ and $(Q, \leq)$ be two posets. We define a binary relation R such that $\forall\ x, y \in P \cup Q$, xRy if and only if $x, y \in P$ and $x \leq_P y$, or $x, y \in Q$ and $x \leq_Q y$. If R is a partial order, then $(P \cup Q, R)$ forms a poset, written as $P \tilde{\cup} Q$. We say that P and Q are *(orderly) unionable*. $P \tilde{\cup} Q$ (if exists) is called the *order union* of P and Q.

Note that R may not always be a partial order. For example, let $x, y \in P \cup Q$, $x <_P y$, and $y <_Q x$. Therefore, $(x, y) \in R$ and $(y, x) \in R$. So R is not a partial order. In this case, we say that $P\tilde{\cup}Q$ does not exist.

If P and Q are disjoint, then $P\tilde{\cup}Q$ always exists.

The $\tilde{\cup}$ operation is commutative, idempotent and associative. I.e.,

Property 2.6 $\forall$ *posets* P, Q, *and* S,

1. *(Commutativity)* $P\tilde{\cup}Q$ *exists* $\Leftrightarrow$ $Q\tilde{\cup}P$ *exists* $\Rightarrow$ $P\tilde{\cup}Q = Q\tilde{\cup}P$.

2. *(Idempontence)* $P\tilde{\cup}P$ *always exists, and* $P\tilde{\cup}P = P$.

3. *(Associativity)*

   $(P\tilde{\cup}Q)\tilde{\cup}S$ *exists* $\Leftrightarrow$ $P\tilde{\cup}(Q\tilde{\cup}S)$ *exists* $\Rightarrow$ $(P\tilde{\cup}Q)\tilde{\cup}S = P\tilde{\cup}(Q\tilde{\cup}S)$.

Proof. It is easy to see commutativity and idempotence hold. I will now prove associativity. I first prove the forward implication.

Fact 1: If $(P\tilde{\cup}Q)\tilde{\cup}S$ exists, this implies $P\tilde{\cup}Q$ exists. I.e., $\forall\ x, y \in (P \cup Q)$, $x \leq_{(P\tilde{\cup}Q)} y$ if and only if $x, y \in P$ and $x \leq_P y$, or $x, y \in Q$ and $x \leq_Q y$.

Fact 2: If $(P\tilde{\cup}Q)\tilde{\cup}S$ exists, then $\forall\ x, y \in P \cup Q \cup S$, $x \leq_{((P\tilde{\cup}Q)\tilde{\cup}S)} y$ if and only if $x, y \in (P \cup Q)$ and $x \leq_{(P\tilde{\cup}Q)} y$, or $x, y \in S$ and $x \leq_S y$.

We now define a binary relation R on $P \cup (Q \cup S)$. $\forall\ x, y \in P \cup (Q \cup S)$, xRy if

and only if one of the following holds:

$$x, y \in P \text{ and } x \leq_P y, \tag{2.1}$$

$$x, y \in Q \text{ and } x \leq_Q y, \tag{2.2}$$

$$x, y \in S \text{ and } x \leq_S y. \tag{2.3}$$

Combining Fact 1 and 2, we know that R is a partial order.

We further define a binary relation R' such that $\forall\ x, y \in Q \cup S$, $xR'y$ if and only if either (2.2) or (2.3) holds. R' is a partial order induced from R. Therefore, $Q\tilde{\cup}S$ exists.

Now we define a binary relation R'' such that $\forall\ x, y \in P \cup (Q\tilde{\cup}S)$, $xR''y$ if and only if either (2.1) holds, or $x, y \in (Q\tilde{\cup}S)$ and $x \leq_{(Q\tilde{\cup}S)} y$. It is easy to see that R'' is equivalent to R. Thus $P\tilde{\cup}(Q\tilde{\cup}S)$ exists, and $(P\tilde{\cup}Q)\tilde{\cup}S = P\tilde{\cup}(Q\tilde{\cup}S)$.

This completes the forward implication. The backward implication can be easily proved by using the forward one and commutativity of the $\tilde{\cup}$ operation.

$$\begin{aligned}
P\tilde{\cup}(Q\tilde{\cup}S) &= (Q\tilde{\cup}S)\tilde{\cup}P \quad \text{(commutativity)} \\
&= (S\tilde{\cup}Q)\tilde{\cup}P \quad \text{(commutativity)} \\
&= S\tilde{\cup}(Q\tilde{\cup}P) \quad \text{(forward implication)} \\
&= S\tilde{\cup}(P\tilde{\cup}Q) \quad \text{(commutativity)} \\
&= (P\tilde{\cup}Q)\tilde{\cup}S. \quad \text{(commutativity)}
\end{aligned}$$

Thus the backward implication also holds. □

The identity element of the $\tilde{\cup}$ operation is the empty set $\emptyset$ (which is a poset). I.e., $\forall$ posets P, $P\tilde{\cup}\emptyset$ always exists, and $P\tilde{\cup}\emptyset = P$.

2.1.2 Diagram of Posets

Definition 2.7 (Covering Relation) *Let P be a poset and let $x, y, z \in P$. We say that x is* ***covered by*** *y if $x < y$ and $x \leq z < y \Rightarrow z = x$. I.e., there exists no element $z \in P$ such that $x < z < y$.*

Intuitively, x is covered by y if y is the immediate successor (in the ascending order) of x. An element $x \in P$ may have more than one immediate successor.

A finite poset $(P, \leq)$ can be represented by a directed acyclic graph. We use a node to represent each element in P. $\forall\ x, y \in P$, if x is covered by y, we add a directed edge that points from x to y. Such graph is called a *diagram*[1] of P. For example, Figure 2.1(a) shows the diagram of the power set of $\{0, 1\}$, which is a poset with the ordering relation of set inclusion $\subseteq$.

It is impossible to represent the whole of an infinite poset by a diagram. However, if the structure is sufficiently regular, it can often be suggested by a diagram. Figure 2.1(b) shows such diagram for the set of natural numbers with the usual ordering relation $\leq$. We will see more examples of diagrams of infinite posets in Chapter 5.

Let $(P, \leq)$ and $(Q, \leq)$ be two posets. We can form the diagram of $P \tilde{\cup} Q$, if it exists, by composing the diagrams of P and Q followed by *transitive reduction*. Transitive reduction [3] removes edges in the composed graph that are not covering relations. Those edges can be inferred due to the transitive property of partial order.

A graphical composition of P and Q that results in directed cycles is not a diagram

[1] This is a variation of the Hasse diagram [23].

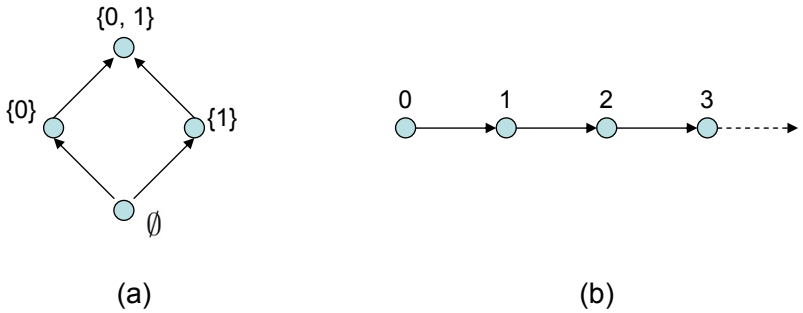


Figure 2.1: Examples of diagrams of posets. Figure (a) is the diagram for the power set of {0, 1}, where the ordering relation is the set inclusion $\subseteq$. Figure (b) is the diagram for the set of natural numbers, where the ordering relation is the usual $\leq$.

of a partial order. In other words, we can tell whether $P \tilde{\cup} Q$ exists by composing the two diagrams and checking for existence of directed cycles. Graphical representation is useful since it is more intuitive. For example, it is not difficult to see, from a diagrammatic perspective, that Property 2.6 holds.

2.2 The Tagged Signal Model

The tagged signal model [49] 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 [6], interaction categories [1], and interaction semantics [65]. 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 [53, 54], 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 $\mathcal{T}$.

Lemma 2.8 *Let $\mathcal{D}(\mathcal{T})$ be the set of all down sets of a poset $\mathcal{T}$.*

1. *$(\mathcal{D}(\mathcal{T}), \subseteq)$ is a CPO.*

2. *$(\mathcal{D}(\mathcal{T}), \subseteq)$ is a complete lattice.*

3. *$(\mathcal{D}(\mathcal{T}), \subseteq)$ is totally ordered if and only if $(\mathcal{T}, \leq)$ is totally ordered.*

PROOF. Part 1 and Part 2 come from [53]. We now prove Part 3.

We first prove the backward implication. Given $\mathcal{T}$ is totally ordered, we need to show that $\forall\ T_1,\ T_2 \in \mathcal{D}(\mathcal{T})$, either $T_1 \subseteq T_2$ or $T_2 \subseteq T_1$. We define a set $T_1 \backslash T_2 = \{t \mid t \in T_1 \text{ and } t \notin T_2\}$. If $T_1 \backslash T_2 = \emptyset$, then $T_1 \subseteq T_2$. Otherwise, there exists at least one tag $t_1 \in T_1 \backslash T_2$. If $\exists\ t_2 \in T_2$ such that $t_1 \leq t_2$, then $t_1 \in T_2$. Hence, since $\mathcal{T}$ is

totally ordered, $t_2 \leq t_1$ for all $t_2 \in T_2$. Therefore, $\forall\ t_2 \in T_2$, $t_2 \in T_1$. This means $T_2 \subseteq T_1$. In summary, we have either $T_1 \subseteq T_2$ or $T_2 \subseteq T_1$. I.e., $\mathcal{D}(\mathcal{T})$ is totally ordered.

We next prove the forward implication. We need to show that $\forall\ t_1, t_2 \in \mathcal{T}$, either $t_1 \leq t_2$ or $t_2 \leq t_1$. Let

$$T_1 = \{t \in \mathcal{T} \mid t \leq t_1\},$$

$$T_2 = \{t \in \mathcal{T} \mid t \leq t_2\}.$$

T_1 and T_2 are two down sets.

Assuming $\mathcal{D}(\mathcal{T})$ is totally ordered, we have either $T_1 \subseteq T_2$ or $T_2 \subseteq T_1$. If $T_1 \subseteq T_2$, since $t_1 \in T_1$, then $t_1 \in T_2$. This leads to $t_1 \leq t_2$. Similarly, if $T_2 \subseteq T_1$, we can prove that $t_2 \leq t_1$. Therefore, $\mathcal{T}$ is totally ordered. $\square$

Part 3 of Lemma 2.8 tells us that the total ordering of a tag set $\mathcal{T}$ is a necessary and sufficient condition for the total ordering of its $\mathcal{D}(\mathcal{T})$. Thus we do not need to make distinction between these two concepts.

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 [53]. 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 $s \in a$; $s \in a$ 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 particularly 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 define a *composite actor*. The composite actor is defined to be the intersection of all possible behaviors of the actors A and the connectors C [49].

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

$$F_a \colon \mathcal{S}^{|I|} \to \mathcal{S}^{|O|}, \tag{2.4}$$

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 Q of external ports, where $Q \bigcap P = \emptyset$ (see Figure 2.2). 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*.

2.3 Syntax

Actor-oriented languages can be either self-contained programming languages (e.g. Esterel, Lustre, LabVIEW) or coordination languages (e.g. Manifold [59], 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

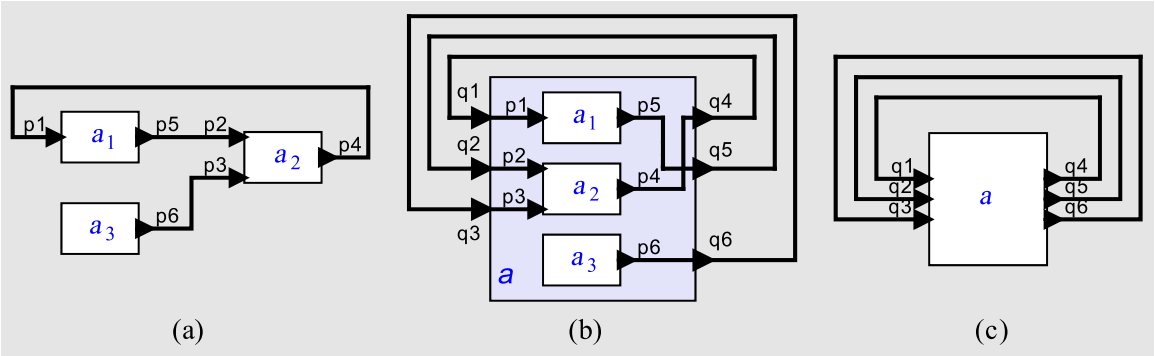


Figure 2.2: A composition of three actors and its interpretation as a feedback system. Figure (b) rearranges the three actors a_1, a_2 and a_3 in Figure (a) and composes them into one composite actor a. Figure (c) is the view from higher level of hierarchy, where details of the composite actor b are hidden. $P = \{p1, p2, p3, p4, p5, p6\}$ 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.

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 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 2.2(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 2.2(a) is closed. In Figure 2.2(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 2.2(c), the internal structure of the composite actor is hidden. Using the techniques introduced in this dissertation, 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 2.2(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 2.2(a), the signal going from $p5$ to $p2$ induces ports $q5$ and $q2$ in Figures 2.2(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 2.2(c).

2.4 Fixed Point Semantics

It is easy to see that if actors a_1, a_2, and a_3 in Figure 2.2(b) are functional, then the composite actor a in Figure 2.2(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 2.2(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 2.2(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. *Monotonicity* and *continuity* are two important properties in ensuring the existence of least fixed point and finding it constructively.

Definition 2.9 *Let $(P, \leq)$ and $(Q, \leq)$ be two posets. A function $f : P \to Q$ is said to be* ***monotonic*** *if $\forall\ x, y \in P$, $x \leq y$ in $P \Rightarrow\ f(x) \leq f(y)$ in Q.*

Definition 2.10 *Let $(P, \leq)$ and $(Q, \leq)$ be two posets. A function $f : P \to Q$ is said to be* ***(Scott) continuous*** *if for all chain $C \subseteq P$,*

$$f(\bigvee C) = \bigvee \{f(c) \mid c \in C\}.$$

Continuity implies monotonicity [23]. The following two theorems are from [23]. They serve as the base of several results given in this dissertation.

Theorem 2.11 (CPO Fixed Point Theorem) *Let P be a CPO with partial order $\leq$ and the least element $\perp$. Let $f : P \rightarrow P$ be a monotonic function. Then f has a least fixed point. If f is continuous, then the least fixed point is given by*

$$\text{fix}(f) = \bigvee \{f^n(\perp) \mid n \in \mathbb{N}\},$$

where $\mathbb{N} = \{0, 1, 2, ...\}$ is the set of natural numbers.

Theorem 2.12 (Knaster-Tarski Fixed Point Theorem) *Let P be a complete lattice with partial order $\leq$. Let $f : P \rightarrow P$ be a monotonic function. The least fixed point of f exists and is given by*

$$\text{fix}(f) = \bigwedge \{x \in P \mid f(x) \leq x\}.$$

Since the set of signals $\mathcal{S}^N$ is a CPO with the prefix order, the least fixed point of F_a is assured of existing if F_a is monotonic, and a constructive procedure exists for finding that least fixed point if F_a is also (Scott) continuous. It is easy to show that if a_1, a_2, and a_3 in Figure 2.2(b) have continuous actor functions, then so does a in Figure 2.2(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 2.2(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 [17] 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.

Chapter 3

Causality Interfaces

In this chapter, I formally present the causality interface theory for functional actors. Functional actors have deterministic behaviors provided their inputs are given. This determinism is desirable in system design in practice. Many actors we use in system design are functional actors, e.g., an adder, an XOR gate, an FFT transformer, and etc. Functional actors (or functions) are better understood by engineers, and have been well studied in the literature. The development of causality interfaces is based on the functional abstraction of actors.

3.1 Dependency Algebra

Through the tags, the tagged signal model represents causality relationship via ordering constraints on the tags. A similar representation is accomplished in the reactive modules model [4]. However, this representation of causality is low-level, and

difficult to abstract. This dissertation gives an algebraic framework that abstracts these causality properties and provides a compositional formalism.

In this section, I introduce the *dependency algebra* $(D, \leq, \oplus, \otimes)$. The dependency set D is a poset 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,

$$\forall\ d_1, d_2, d_3 \in D, \quad (d_1 \oplus d_2) \oplus d_3 = d_1 \oplus (d_2 \oplus d_3), \tag{3.1}$$

$$\forall\ d_1, d_2, d_3 \in D, \quad (d_1 \otimes d_2) \otimes d_3 = d_1 \otimes (d_2 \otimes d_3). \tag{3.2}$$

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

$$\forall\ d_1, d_2 \in D, \quad d_1 \oplus d_2 = d_2 \oplus d_1, \tag{3.3}$$

and idempotent,

$$\forall\ d \in D, \quad d \oplus d = d. \tag{3.4}$$

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

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

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

$$\forall\ d \in D, \quad d \otimes \mathbf{0} = \mathbf{0}.$$

The ordering relation $\leq$ on the set D is a partial order.

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

$$\forall\ d_1, d_2 \in D, \quad d_1 \oplus d_2 \leq d_1. \tag{3.5}$$

Using these axioms, we get the following property:

Property 3.1 *The additive identity* $\mathbf{0}$ *is the top element of the partial order* $(D, \leq)$.

PROOF. Using (3.5), let $d_1 = \mathbf{0}$, from which we conclude

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

□

3.2 Basic Definition of Causality Interfaces

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

$$\delta_a \colon P_i \times P_o \to D, \tag{3.6}$$

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 $\mathbf{0}$.

The choice of dependency algebra depends on the model of computation. In the next section, I will present a dependency algebra for functional actors. This algebra model is similar in spirit to the tagged signal model. The purpose is not to invent

a "grand unified" dependency algebra for *all* models of computation. As we will see later, for certain cases, many representations of dependencies take much simpler forms.

3.3 Causality Interfaces for Functional Actors

A functional actor can be viewed as a function that maps its input signals to its output signals. We define the dependency set D to be a set of functions:

$$D = (\mathcal{D}(\mathcal{T}) \rightarrow \mathcal{D}(\mathcal{T})), \tag{3.7}$$

where $(X \rightarrow Y)$ denotes the set of total functions with domain X and range contained by Y, and $\mathcal{D}(\mathcal{T})$, as before, 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 $(d_1 \oplus d_2)\colon \mathcal{D}(\mathcal{T}) \rightarrow \mathcal{D}(\mathcal{T})$ is defined by

$$\forall\ T \in \mathcal{D}(\mathcal{T}), \quad (d_1 \oplus d_2)(T) = d_1(T) \cap d_2(T). \tag{3.8}$$

To see that (3.8) 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 (3.7) satisfies all of the axioms described in Section 3.1.

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 2.2, 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 $\mathbf{1} = d_I$.

3.3.1 Liveness, Monotonicity and Continuity

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.

The notion of monotonicity and Scott continuity we reviewed in Chapter 2 can be easily adapted to causality interfaces. Recall that 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 3.2 *Let p be an input port and p' be an output port of a monotonic actor a. Then $\delta_a(p, p')$ is monotonic.*

Property 3.3 $\forall$ *monotonic functions $d_1, d_2, d_3 \in D$, where $D = (\mathcal{D}(\mathcal{T}) \to \mathcal{D}(\mathcal{T}))$,*

$$d_1 \otimes (d_2 \oplus d_3) = (d_1 \otimes d_2) \oplus (d_1 \otimes d_3) \tag{3.9}$$

$$(d_2 \oplus d_3) \otimes d_1 \leq (d_2 \otimes d_1) \oplus (d_3 \otimes d_1). \tag{3.10}$$

If $\mathcal{T}$ is totally ordered, then

$$(d_2 \oplus d_3) \otimes d_1 = (d_2 \otimes d_1) \oplus (d_3 \otimes d_1). \tag{3.11}$$

If (3.9) is satisfied, then $(\oplus, \otimes)$ is said to be *left-distributive* on the subset of monotonic functions in D. Similarly, if (3.11) is satisfied, $(\oplus, \otimes)$ is said to be *right-distributive*. We say that $(\oplus, \otimes)$ is *distributive* if it is both left and right distributive. Distributivity on the subset of monotonic functions where the tag set $\mathcal{T}$ is totally ordered will be used to show that causality analysis for intersecting cycles can be performed independently and in parallel.

PROOF. We first prove (3.9). $\forall\ T \in \mathcal{D}(\mathcal{T})$,

$$\begin{aligned}
& (d_1 \otimes (d_2 \oplus d_3))(T) \\
= \ & (d_2 \oplus d_3)(d_1(T)) \\
= \ & (d_2(d_1(T))) \cap (d_3(d_1(T))) \\
= \ & (d_1 \otimes d_2)(T) \cap (d_1 \otimes d_3)(T) \\
= \ & ((d_1 \otimes d_2) \oplus (d_1 \otimes d_3))(T).
\end{aligned}$$

We next prove (3.10). $\forall\ T \in \mathcal{D}(\mathcal{T})$,

$$((d_2 \oplus d_3) \otimes d_1)(T) = d_1(d_2(T) \cap d_3(T)), \tag{3.12}$$

$$((d_2 \otimes d_1) \oplus (d_3 \otimes d_1))(T) = d_1(d_2(T)) \cap d_1(d_3(T)). \tag{3.13}$$

Note that $(d_2(T) \cap d_3(T)) \subseteq d_2(T)$. Since d_1 is monotonic, we have $d_1(d_2(T) \cap d_3(T)) \subseteq d_1(d_2(T))$. Similarly, $d_1(d_2(T) \cap d_3(T)) \subseteq d_1(d_3(T))$. Therefore, $d_1(d_2(T) \cap d_3(T)) \subseteq (d_1(d_2(T)) \cap d_1(d_3(T))) = ((d_2 \otimes d_1) \oplus (d_3 \otimes d_1))(T)$. Hence, (3.10) holds.

We next prove (3.11). If $\mathcal{T}$ is totally ordered, then $\mathcal{D}(\mathcal{T})$ is also totally ordered. Therefore, either $d_2(T) \subseteq d_3(T)$ or $d_3(T) \subseteq d_2(T)$. Without loss of generality, we assume $d_2(T) \subseteq d_3(T)$. Since d_1 is monotonic, then $d_1(d_2(T)) \subseteq d_1(d_3(T))$. Therefore,

$$d_1(d_2(T) \cap d_3(T)) = d_1(d_2(T)) = d_1(d_2(T)) \cap d_1(d_3(T)),$$

and (3.11) holds. □

In this dissertation, I focus on actors that are (Scott) continuous. (Scott) continuity is a stronger property than monotonicity. Recall that 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 3.4 *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 [23], so it follows that the causality interfaces of a continuous actor are also monotonic.

3.3.2 The $\prec$ Relation on Dependency Algebra

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

Property 3.5 *The relation $\prec$ is a strict partial order. I.e., 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* $d_2 \prec d_3 \Rightarrow d_1 \prec d_3$.

PROOF. It is easy to see from the first requirement of the definition that irreflexivity holds for the $\prec$ relation.

Note that based on the second requirement of the definition, $\forall\ d_1, d_2 \in D$, $d_1 \prec d_2\ \Rightarrow\ \forall\ T \in \mathcal{D}(\mathcal{T}),\ d_1(T) \subseteq d_2(T)$.

To prove antisymmetry, consider two functions $d_1, d_2 \in D$. If $d_1 \prec d_2$, then $\exists\ T_0 \in \mathcal{D}(\mathcal{T})$ such that $d_1(T_0) \subset d_2(T_0)$. (Otherwise, $\forall\ T \in \mathcal{D}(\mathcal{T})$, $d_1(T) = d_2(T)$. Then $d_1 = d_2$. This contradicts with the fact that $d_1 \prec d_2$.) Therefore, $d_2 \not\prec d_1$.

Finally, we prove transitivity. We define a subset $\mathcal{A}$ of $\mathcal{D}(\mathcal{T})$ such that $\mathcal{A} = \{T \mid T \in \mathcal{D}(\mathcal{T}),\ d_1(T) \subset d_2(T)\}$. Since $d_1 \prec d_2$, then there exists at least one $T_0 \in \mathcal{D}(\mathcal{T})$ such that $d_1(T_0) \subset d_2(T_0)$. Therefore $\mathcal{A}$ is a non-empty set.

$\forall\ T \in \mathcal{D}(\mathcal{T})$,

1. If $T \in \mathcal{A}$, we have $d_1(T) \subset d_2(T) \subseteq d_3(T)$ (because $d_2 \prec d_3$). Thus $d_1(T) \subset d_3(T)$. Since $\mathcal{A}$ is a non-empty set, then $d_1 \neq d_3$.

2. If $T \notin \mathcal{A}$, then $d_1(T) = d_2(T) = \mathcal{T}$. Since $d_2(T) \subseteq d_3(T)$, then $d_1(T) = d_3(T) = \mathcal{T}$.

Thus $\forall\ T \in \mathcal{D}(\mathcal{T})$, $d_1(T) \subset d_3(T)$ or $d_1(T) = d_3(T) = \mathcal{T}$, and $d_1 \neq d_3$. This concludes $d_1 \prec d_3$. $\square$

Property 3.6 $\forall\ d_1, d_2, d_3 \in D$, *where* $D = (\mathcal{D}(\mathcal{T}) \to \mathcal{D}(\mathcal{T}))$,

$$d_1 \prec d_2 \oplus d_3 \quad \Rightarrow \quad d_1 \prec d_2 \text{ and } d_1 \prec d_3. \tag{3.14}$$

If $\mathcal{T}$ *is totally ordered, then*

$$d_1 \prec d_2 \oplus d_3 \quad \Leftrightarrow \quad d_1 \prec d_2 \text{ and } d_1 \prec d_3. \tag{3.15}$$

PROOF. We define a subset $\mathcal{A}$ of $\mathcal{D}(\mathcal{T})$ such that $\mathcal{A} = \{T \mid T \in \mathcal{D}(\mathcal{T}) \text{ and } d_1(T) \neq \mathcal{T}\}$. We first prove (3.14). Assuming $d_1 \prec d_2 \oplus d_3$, $d_1 \neq d_\top$, where $d_\top$ is the top element of D, so $\mathcal{A}$ is not empty. Then $\forall\ T \in \mathcal{D}(\mathcal{T})$,

1. if $T \in \mathcal{A}$, then $d_1(T) \subset (d_2 \oplus d_3)(T) \subseteq d_2(T)$. I.e., $d_1(T) \subset d_2(T)$. And since $\mathcal{A}$ is not empty, then $d_1 \neq d_2$.

2. if $T \notin \mathcal{A}$, then $d_1(T) = (d_2 \oplus d_3)(T) = \mathcal{T}$. Therefore, $d_2(T) = \mathcal{T}$.

In summary, $d_1 \neq d_2$ and $\forall\ T \in \mathcal{D}(\mathcal{T})$, $d_1(T) \subset d_2(T)$ or $d_1(T) = d_2(T) = \mathcal{T}$. I.e., $d_1 \prec d_2$. Similarly, we can prove that $d_1 \prec d_3$.

The forward implication of (3.15) is proven in (3.14). Assuming that $\mathcal{T}$ is totally ordered, we now prove the backward implication. Assuming $d_1 \prec d_2$ and $d_1 \prec d_3$, $d_1 \neq d_\top$, so $\mathcal{A}$ is not empty. Then $\forall\ T \in \mathcal{D}(\mathcal{T})$,

1. if $T \in \mathcal{A}$, then $d_1(T) \subset d_2(T)$ and $d_1(T) \subset d_3(T)$. Since $\mathcal{T}$ is totally ordered, so is $\mathcal{D}(\mathcal{T})$. Thus we have either $d_2(T) \subseteq d_3(T)$ or $d_3(T) \subseteq d_2(T)$. Without loss of generality, we assume $d_2(T) \subseteq d_3(T)$. Therefore, $(d_2 \oplus d_3)(T) = d_2(T)$. This leads to $d_1(T) \subset (d_2 \oplus d_3)(T)$. And since $\mathcal{A}$ is not empty, $d_1 \neq d_2 \oplus d_3$.

2. if $T \notin \mathcal{A}$, then $d_1(T) = d_2(T) = d_3(T) = \mathcal{T}$. I.e., $d_1(T) = (d_2 \oplus d_3)(T) = \mathcal{T}$.

In summary, $d_1 \neq d_2 \oplus d_3$ and $\forall\ T \in \mathcal{D}(\mathcal{T})$, $d_1(T) \subset (d_2 \oplus d_3)(T)$ or $d_1(T) = (d_2 \oplus d_3)(T) = \mathcal{T}$. I.e., $d_1 \prec d_2 \oplus d_3$. □

The following example shows that the backward implication of (3.15) does not hold if $\mathcal{T}$ is not totally ordered.

Example 3.7 Consider the following tag set $\mathcal{T} = \{a, \alpha\}$, where a and α are not comparable. Then,

$$\mathcal{D}(\mathcal{T}) = \{\emptyset, \{a\}, \{\alpha\}, \{a, \alpha\}\}.$$

We define d_1 to be the identity function d_I, and $d_2,\ d_3$ are given as follows:

$$d_2(\emptyset) = \{a\}, \quad d_3(\emptyset) = \{\alpha\},$$
$$\forall\ T \in \mathcal{D}(\mathcal{T}), \quad T \neq \emptyset \ \Rightarrow\ d_2(T) = d_3(T) = \{a, \alpha\}.$$

Therefore, we have $d_1 \prec d_2$ and $d_1 \prec d_3$. However,

$$(d_2 \oplus d_3)(\emptyset) = d_2(\emptyset) \cap d_3(\emptyset) = \{a\} \cap \{\alpha\} = \emptyset = d_1(\emptyset).$$

Thus $d_1 \not\prec d_2 \oplus d_3$. We also note that in this case the least fixed point of $(d_2 \oplus d_3)$ is $\emptyset$. □

Property 3.6 can be easily extended to $\oplus$ operations on arbitrary finite number of dependencies in D.

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

Theorem 3.8 *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. Since continuity implies monotonicity, part (1) can be proven directly from the Knaster-Tarski fixed point theorem.

Part (2): If $d_I \prec d$, then $\forall\ T \in \mathcal{D}(\mathcal{T})$, $d_I(T) \subset d(T)$ or $d_I(T) = d(T) = \mathcal{T}$.

1. If $T \neq \mathcal{T}$, then $d_I(T) = T \neq \mathcal{T}$. Therefore, $T = d_I(T) \subset d(T)$.

2. If $T = \mathcal{T}$, since $d_I \prec d$, $d_I(\mathcal{T}) \subseteq d(\mathcal{T})$. Note that $\mathcal{T}$ is the top element of $\mathcal{D}(\mathcal{T})$. Therefore, $d(\mathcal{T}) \subseteq \mathcal{T} = d_I(\mathcal{T})$. Thus, $d(\mathcal{T}) = d_I(\mathcal{T}) = \mathcal{T}$.

Therefore, $T_0 = \bigwedge\{T \in \mathcal{D}(\mathcal{T}) \mid d(T) \subseteq T\} = \mathcal{T}$. □

The following example shows that if $\mathcal{T}$ is not totally ordered, then $d_I \prec d$ is not a necessary condition for the least fixed point of d to be $\mathcal{T}$.

Example 3.9 Consider a tag set $\mathcal{T} = \{a, b, \alpha\}$, where $a < b$ and α is neither comparable to a nor to b. The set of all down sets is

$$\mathcal{D}(\mathcal{T}) = \{\emptyset, \{a\}, \{\alpha\}, \{a, b\}, \{a, \alpha\}, \{a, b, \alpha\}\}.$$

We define a function $d : \mathcal{D}(\mathcal{T}) \to \mathcal{D}(\mathcal{T})$ such that

$$d(T) = \begin{cases} \{a, b\}, & \text{if } T = \emptyset,\ \{a\} \text{ or } \{\alpha\} \\ \{a, b, \alpha\}, & \text{otherwise.} \end{cases} \tag{3.16}$$

It is easy to verify that d is a continuous function, and it has a least fixed point $\mathcal{T} = \{a, b, \alpha\}$. However, $d_I \not\prec d$, since $d(\{\alpha\}) = \{a, b\}$, but $\{\alpha\} \not\subset \{a, b\}$. $\square$

Corollary 3.10 *If $d : \mathcal{D}(\mathcal{T}) \to \mathcal{D}(\mathcal{T})$ is a continuous function, where $\mathcal{T}$ 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 3.8. 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 $d(\mathcal{T}) = \mathcal{T} = d_I(\mathcal{T})$, and $\forall\ T \neq \mathcal{T}$, $d(T) \not\subseteq T$. Since $\mathcal{T}$ is totally ordered, and hence $\mathcal{D}(\mathcal{T})$ is also totally ordered, $d_I(T) = T \subset d(T)$ for all $T \neq \mathcal{T}$. Therefore, $d_I \prec d$. $\square$

3.4 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. We use a node to represent each port. A directed edge is drawn from each input port to each output port of an actor (connector). The weight of each edge (p, p') is the causality interface $\delta(p, p')$, which represents the dependency of port p' on p. Given a path $(p_1, p_2, ..., p_n)$, where p_i's $(1 \leq i \leq n)$ are ports of the composition, we define the *gain* of the path to be

$$g_{path} = \delta(p_1, p_2) \otimes \delta(p_2, p_3) \otimes ... \otimes \delta(p_{n-1}, p_n).$$

We will first discuss feedforward compositions and then deal with feedback compositions.

3.4.1 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 3.1 shows a feedforward composition, which is abstracted into a single actor b with external input port $q1$ and output port $q2$. The dependency graph of the composition is shown in Figure 3.2. 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

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

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 3.3 shows a slightly more complicated example. Its dependency graph is shown in Figure 3.4. We note that there are two parallel paths from port $p5$ to port

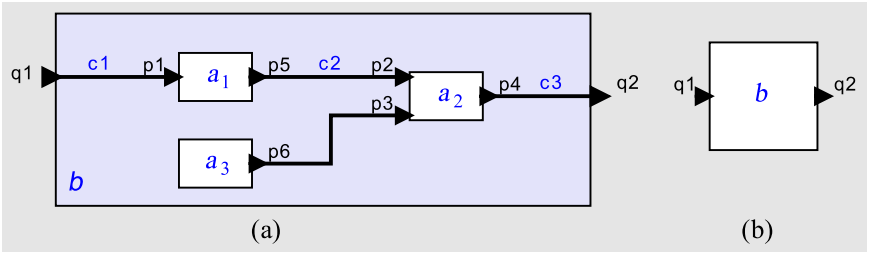


Figure 3.1: A feedforward composition.

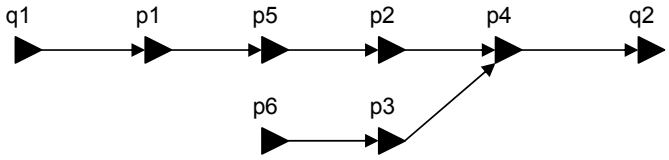


Figure 3.2: The dependency graph of the composition shown in Figure 3.1

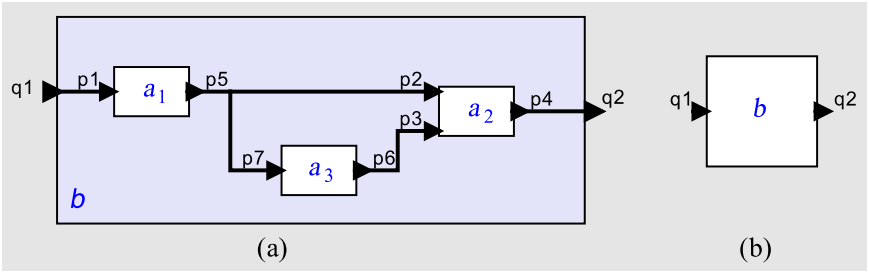


Figure 3.3: A feedforward composition with parallel paths.

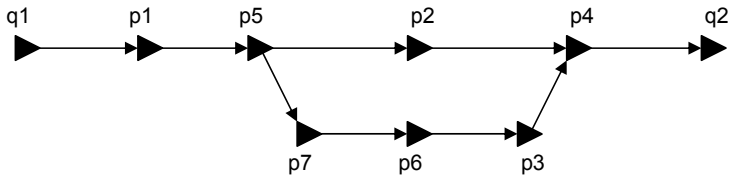


Figure 3.4: The dependency graph of the composition shown in Figure 3.3

$p4$. We get

$$\delta_b(q1, q2) = \delta_{a_1}(p1, p5) \otimes [\delta_{a_2}(p2, p4) \oplus (\delta_{a_3}(p7, p6) \otimes \delta_{a_2}(p3, p4))], \tag{3.17}$$

where we have omitted the causality interfaces for connectors.

3.4.2 Feedback Compositions

The dependency graph of a feedback system contains cyclic paths. Given a cyclic path $c = (p_1, p_2, ..., p_n, p_1)$, $c' = (p_i, ..., p_n, p_1, ..., p_i)$ 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. I choose the least fixed point semantics, since it is used in quite a few models of computation, including timed models, synchronous languages, dataflow models and process networks. Feedback compositions of causality interfaces under other semantics (e.g., the greatest fixed point semantics) can be formulated in similar way.

We now begin by considering simple cases of feedback systems and build up to the general case. Consider the composition shown in Figure 3.5, where actor a is a feedforward composite actor. From Section 3.4.1, we can determine its causality interfaces and we know it is live if its component actors are live.

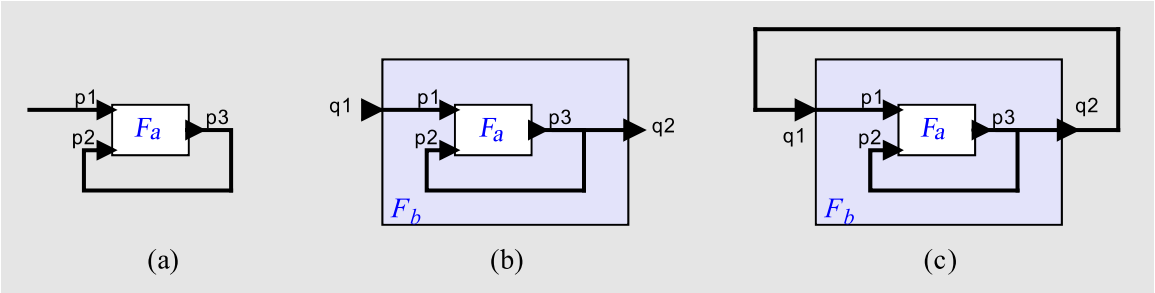


Figure 3.5: An open composition with feedback loops.

The following two lemmas are useful. The first is an adaptation of Lemma 8.10 in [70]:

Lemma 3.11 *Consider two CPOs S_1 and S_2, and a continuous function*

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

For a given $s_1 \in S_1$, we define the function $F_a(s_1) : S_2 \to S_2$ 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 3.5(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 $\bigvee\{(F_a(s_1))^n(\bot) \mid n \in \mathbb{N}\}$ [23].

The second lemma comes from [54]:

Lemma 3.12 *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 3.5(b) defines a continuous function F_b from port $q1$ to port $q2$.

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,

$$\begin{aligned} \delta_b(q1, q2)(T_1) = \mathrm{dom}(F_b(s_1)) &= \mathrm{dom}(\bigvee\{(F_a(s_1))^n(\perp) \mid n \in \mathbb{N}\}) \\ &= \bigvee\{\mathrm{dom}((F_a(s_1))^n(\perp)) \mid n \in \mathbb{N}\} \\ &= \bigvee\{(f_a(T_1))^n(\emptyset) \mid n \in \mathbb{N}\} \end{aligned} \tag{3.18}$$

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

Given that actor a is live, $\delta_a(p1, p3)(\mathcal{T}) = \mathcal{T}$. Therefore, $f_a(\mathcal{T}) = \delta_a(p2, p3)$. If $\mathbf{1} \prec \delta_a(p2, p3)$, where $\mathbf{1} = d_I$ is the multiplicative identity, then the least fixed point of $f_a(\mathcal{T})$ is $\mathcal{T}$ (due to Theorem 3.8). Hence actor b is live.

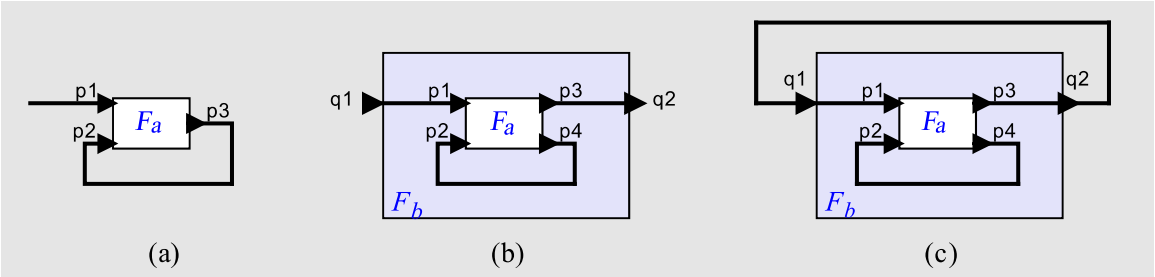


Figure 3.6: An open composition with feedback loops that has the structure of Figure 2.2.

Given the causality interface for actor b, as shown in (3.18), we now form the nested feedback composition of Figure 3.5(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 2.2, we add an additional output port to actor a in Figure 3.6. We can easily adapt Lemmas 3.11 and 3.12 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 2.2(c), we can determine from the causality interfaces of a, whether a composition is live.

If $\mathcal{T}$ is totally ordered, the following lemma helps us to give the causality interface of feedback composition in a much simpler form than (3.18).

Lemma 3.13 *Consider a continuous function $\delta : \mathcal{D}(\mathcal{T}) \to \mathcal{D}(\mathcal{T})$, where $\mathcal{T}$ 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 that $\mathcal{T}$ is totally ordered $\Leftrightarrow$ $\mathcal{D}(\mathcal{T})$ 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 3.14 *Given the composite actor b as shown in Figure 3.5(b), and assuming $\mathcal{T}$ is totally ordered,*

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 3.13.

Part (2): We first prove the backward implication. Assuming $\mathbf{1} \prec \delta_a(p2, p3)$, $T_0 = \mathcal{T}$. Therefore $\delta_b(q1, q2) = \delta_a(p1, p3)$. Thus b is live since a is live.

We next prove the forward implication. Assuming b is live, $\delta_b(q1, q2)(\mathcal{T}) = \delta_a(p1, p3)(\mathcal{T}) \cap T_0 = \mathcal{T}$. Therefore $T_0 = \mathcal{T}$. Due to Corollary 3.10, 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. □

Due to Corollary 3.14, $\delta_b(q1, q2) \leq \delta_a(p1, p3)$. I.e., the dependency of $q2$ on $q1$ is at least as strict as if there were no feedback connection from port $p3$ to $p2$. The equality holds if and only if the feedback connection does not result in deadlock.

In the case $\mathcal{T}$ is totally ordered, the verification algorithm can be summarized as follows. First we convert any actor network of interest into one actor a with N feedback connections. (We have given the procedure to do so in Section 2.3). Let the input ports of a be $i_1, i_2, ..., i_N$, and the output ports be $o_1, o_2, ..., o_N$, where $\forall\ j \in \mathbb{N}$, $1 \leq j \leq N$, i_j and o_j are connected by a connector c_j, as shown in Figure 3.7(a). The causality interfaces of a are computed using the feedforward composition procedure described in Section 3.4.1.

Figures 3.7(b) and (c) show the procedure for reducing a system with n feedback loops to one with $n - 1$ feedback loops. If $\mathbf{1} \prec \delta_a(i_n, o_n)$, then actor a' is live. Due to

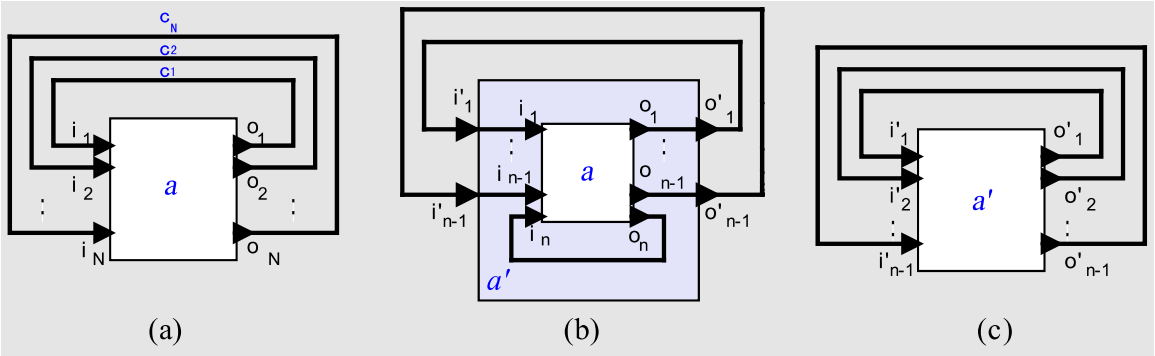


Figure 3.7: A system with n feedback connections is reduced to a system with $n-1$ feedback connections.

Corollary 3.14, the causality interfaces of a' are given by

$$\forall\ j,k \in \mathbb{N}, \quad 1 \leq j,k \leq n-1,$$

$$\delta_{a'}(i'_j, o'_k) = \delta_a(i_j, o_k) \oplus [\delta_a(i_j, o_n) \otimes \delta_a(i_n, o_k)]. \tag{3.19}$$

Therefore, we can verify such a network with N feedback loops using the following steps.

[**Verification Algorithm for Liveness**]:

Let $n = N$.

1. If $n = 0$, declare the network to be live. Exit.

2. If $\mathbf{1} \not\preceq \delta_a(i_n, o_n)$, declare the network not to be live. Exit.

3. $\forall\ 1 \leq j,k \leq n-1$, replace the causality interface $\delta_a(i_j, o_k)$ with what is given in (3.19). Let $n = n-1$. Go to step 1.

In the case $\mathcal{T}$ is not totally ordered, the algorithm is similar. However, in step 2, if $\mathbf{1} \not\preceq \delta_a(i_n, o_n)$, we shall exit the algorithm by claiming we cannot use the causality

interface approach to verify the liveness of the network. This is because $\mathbf{1} \prec \delta_a(i_n, o_n)$ is only a sufficient but not necessary condition. And in step 3, the new causality interfaces are not as simple as given in (3.19). One has to construct the least fixed point using (3.18).

Correctness of the algorithm.

1. If an execution of the algorithm exits in step 2, then due to Corollary 3.14, the signal in connector c_n (connecting port i_n and o_n) is not complete. Thus the network is not live.

2. (a) If an execution of the algorithm exits in step 1, then the signal in connector c_1 is complete.

   (b) Given any $k \in \mathbb{N}$ where $1 \leq k \leq N$, for all $j \in \mathbb{N}$ where $1 \leq j < k$, if the signal in connector c_j is complete, then the signal in connector c_k is complete. This is the definition of liveness for open compositions. (See Figure 3.7(b). The feedback connection inside a' has a complete signal if its input signals are complete.)

   By induction, we know that signals in all N connectors are complete. Therefore, the actor network is live.

Complexity of the algorithm.

The number of feedback loops N is bounded by the number of connectors in an actor network. The above algorithm has at most N iterations, since in step 3, we

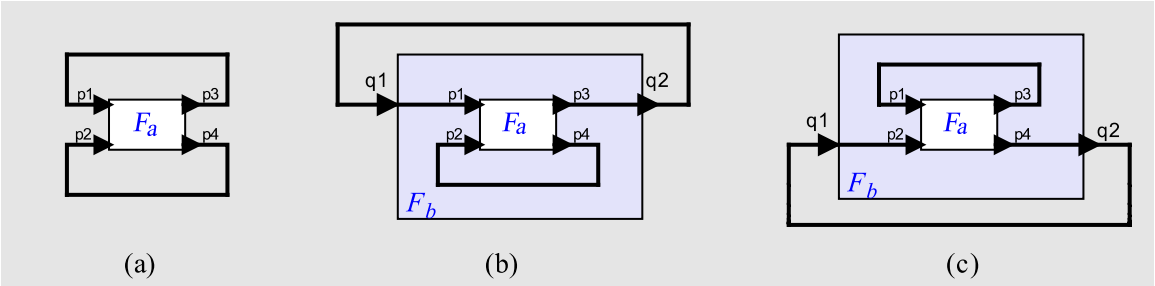


Figure 3.8: A composition with two feedback loops. Figures (b) and (c) show two different ways to verify whether this composition is live.

replace n with $n-1$. The i-th iteration consists of one verification of $\mathbf{1} \prec \delta$ and $(N-i)^2$ computations of the new causality interfaces, where each new interface consists of one $\oplus$ operation and one $\otimes$ operation (see (3.19)). Therefore, the overall computation includes at most N verifications of $\mathbf{1} \prec \delta$ and $\sum_{n=1}^{N}(n-1)^2 = \frac{1}{6}N(N-1)(2N-1)$ number of $\oplus$ and $\otimes$ operations. I.e., the verification consists of $O(N^3)$ $\oplus$ and $\otimes$ operations, and $O(N)$ verifications of $\mathbf{1} \prec \delta$, where N is the number of connectors in the actor network.

3.5 Discussion

The total ordering of $\mathcal{T}$ leads to a simple form of causality interfaces in feedback compositions, and a necessary and sufficient condition for liveness (so the condition is tight). In this section, I further focus on totally ordered tag sets and give an alternative liveness condition simpler than the verification algorithm in Section 3.4.2.

We consider the composition in Figure 3.8(a). There are two feedback connections, one from port $p4$ to $p2$, the other from $p3$ to $p1$. In Figure 3.8(b), we first consider

the feedback from $p4$ to $p2$. Following the verification algorithm in Section 3.4.2, we know that this network is live if and only if

$$\mathbf{1} \prec \delta_a(p2, p4), \quad \text{and,} \tag{3.20}$$

$$\mathbf{1} \prec \delta_a(p1, p3) \oplus [\delta_a(p1, p4) \otimes \delta_a(p2, p3)]. \tag{3.21}$$

Since $\mathcal{T}$ is totally ordered, due to Property 3.6, (3.21) holds if and only if

$$\mathbf{1} \prec \delta_a(p1, p3), \quad \text{and,} \tag{3.22}$$

$$\mathbf{1} \prec \delta_a(p1, p4) \otimes \delta_a(p2, p3). \tag{3.23}$$

In other words, we are considering three cyclic paths, namely $c_1 = (p2, p4, p2)$, $c_2 = (p1, p3, p1)$ and $c_3 = (p1, p4, p2, p3, p1)$. The composition in Figure 3.8(a) is live if and only if

$$\mathbf{1} \prec g_{c_1} \text{ and } \mathbf{1} \prec g_{c_2} \text{ and } \mathbf{1} \prec g_{c_3}. \tag{3.24}$$

Alternatively, we now verify liveness by first considering the feedback connection from $p3$ to $p1$, as shown in Figure 3.8(c). Eventually, we are considering three cyclic paths: c_1, c_2 (same as above), and $c_3' = (p2, p3, p1, p4, p2)$. The composition is live if and only if

$$\mathbf{1} \prec g_{c_1} \text{ and } \mathbf{1} \prec g_{c_2} \text{ and } \mathbf{1} \prec g_{c_3'}. \tag{3.25}$$

A question that must be answered is whether (3.24) and (3.25) are equivalent. This reduces to the question whether $(\mathbf{1} \prec g_{c_3}) \Leftrightarrow (\mathbf{1} \prec g_{c_3'})$.

Note that c_3 and c_3' are two cyclic paths of the same cycle. Since commutativity

does not hold for the $\otimes$ operator, $g_{c_3} \neq g_{c'_3}$ in general. However, if the tag set $\mathcal{T}$ is totally ordered, we have the following lemma:

Lemma 3.15 *Let $\delta_1, \delta_2 \in (\mathcal{D}(\mathcal{T}) \to \mathcal{D}(\mathcal{T}))$ be two continuous functions, where $\mathcal{T}$ 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. To prove the backward implication, we can prove $\mathbf{1} \not\prec \delta_1 \otimes \delta_2 \;\Rightarrow\; \mathbf{1} \not\prec \delta_2 \otimes \delta_1$. Assuming $\mathbf{1} \not\prec \delta_1 \otimes \delta_2$, there exists a down set T_0 where $T_0 \neq \mathcal{T}$ and $T_0 \not\subset \delta_2(\delta_1(T_0))$. Since $\mathcal{T}$ is totally ordered and hence so is $\mathcal{D}(\mathcal{T})$, this means $\delta_2(\delta_1(T_0)) \subseteq T_0$. From this we can infer that $\delta_1(T_0) \neq \mathcal{T}$ because otherwise $\delta_2(\mathcal{T}) \subseteq T_0 \subset \mathcal{T}$. This contradicts the fact that δ_2 satisfies the liveness condition, i.e., $\delta_2(\mathcal{T}) = \mathcal{T}$.

Since δ_1 is continuous and therefore monotonic,

$$\delta_1(\delta_2(\delta_1(T_0)) \subseteq \delta_1(T_0). \tag{3.26}$$

Since $\delta_1(T_0) \neq \mathcal{T}$, $\mathbf{1} \not\prec \delta_2 \otimes \delta_1$. The forward implication is proved identically. □

Corollary 3.16 *Consider a finite network of continuous and live actors where the tag set $\mathcal{T}$ is totally ordered. Suppose c and c' are two cyclic paths of the same cycle in the dependency graph. Then $\mathbf{1} \prec g_c \;\Leftrightarrow\; \mathbf{1} \prec g_{c'}$.*

PROOF. We consider two cyclic paths $c = (p_1, p_2, ..., p_n, p_1)$ and $c' = (p_i, ..., p_n, p_1, ..., p_i)$ of the same cycle. Since all the actors are live, the causality interfaces

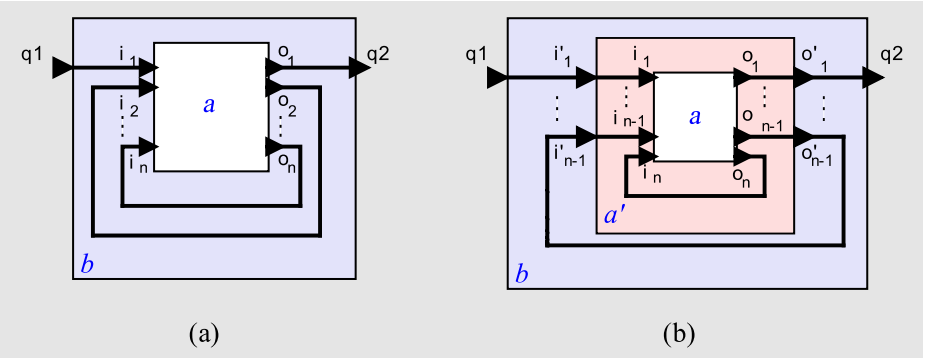


Figure 3.9: An open system with $n-1$ feedback loops.

in the paths satisfy the liveness condition. 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 live, and,

$$g_c = d_1 \otimes d_2$$

$$g_{c'} = d_2 \otimes d_1.$$

Due to Lemma 3.15, we have $\mathbf{1} \prec g_c \Leftrightarrow \mathbf{1} \prec g_{c'}$. $\square$

Therefore, the two ways of checking feedback loops in Figure 3.8 are equivalent. Given a system with N feedback loops, we can check these loops in arbitrary order.

The following lemma will be used to prove Theorem 3.18 below.

Lemma 3.17 *Consider the open system with $n-1$ feedback connections in Figure 3.9(a). Assume actor a is continuous. $\delta_b(q1, q2)$ is the causality interface of the composite actor b. Consider a path $p = (i_1, o_n, i_n, o_{n-1}, ..., i_3, o_2, i_2, o_1)$ in the dependency graph. Then $\delta_b(q1, q2) \leq g_p$.*

PROOF. We prove by induction.

1. If $n = 1$, then there is no feedback connection, and $p = (i_1, o_1)$ and $\delta_b(q1, q2) = \delta_a(i_1, o_1) = g_p$. The lemma holds.

2. We assume that the lemma holds for $n - 1$. We eliminate port i_n, o_n and the feedback connection between them by using the verification algorithm, as depicted in Figure 3.9(b). The causality interfaces of a' are given by (3.19). In particular, $\forall\ j \in \mathbb{N},\ 1 < j < n$,

$$\begin{aligned} \delta_{a'}(i'_j, o'_{j-1}) &= \delta_a(i_j, o_{j-1}) \oplus [\delta_a(i_j, o_n) \otimes \delta_a(i_n, o_{j-1})] \\ &\leq \delta_a(i_j, o_{j-1}), \end{aligned} \tag{3.27}$$

and,

$$\begin{aligned} \delta_{a'}(i'_1, o'_{n-1}) &= \delta_a(i_1, o_{n-1}) \oplus [\delta_a(i_1, o_n) \otimes \delta_a(i_n, o_{n-1})] \\ &\leq \delta_a(i_1, o_n) \otimes \delta_a(i_n, o_{n-1}). \end{aligned} \tag{3.28}$$

Consider the path $p' = (i'_1, o'_{n-1}, i'_{n-1}, ..., i'_2, o'_1)$. By induction, $\delta_b(q1, q2) \leq g_{p'}$. Due to (3.27) and (3.28), $g_{p'} \leq g_p$. This is because all the causality interfaces are monotonic, and function composition (the $\otimes$ operation) preserves monotonicity. Therefore $\delta_b(q1, q2) \leq g_p$.

In conclusion, this lemma holds for all $n \in \mathbb{N}$. $\square$

Intuitively, Lemma 3.17 says that the causality interface of a pair of external (input, output) ports is always bounded by the gain of a path between them in the dependency graph.

We now get to our main theorem.

Theorem 3.18 *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 simple cyclic path c in the dependency graph, $\mathbf{1} \prec g_c$, where $\mathbf{1} = d_I$ is the multiplicative identity.*

PROOF. We assume the actor network of interest has N connectors. We convert it into one composite actor a with N feedback loops. I.e., all the connectors (signals) are exposed outside of a. Let the input ports of a be $i_1, ..., i_N$, and the output ports be $o_1, ...o_N$. $\forall\ j \in \mathbb{N}$ such that $1 \leq j \leq N$, port o_j is connected back to i_j by connector c_j. We use the verification algorithm to check such feedback system, and show that an execution of the algorithm is equivalent to verifying $\mathbf{1} \prec g_c$ for every simple cyclic path c.

We first prove the backward implication. Consider the verification algorithm described in Section 3.4.2. Each step in the algorithm does not increase connectivity between ports in the dependency graph. Step 3 reduces edges by combining parallel paths. Therefore, when we check $\mathbf{1} \prec \delta_a(i_n, o_n)$ in step 2, we are considering some paths (with possible parallel subpaths in the middle) that connect port i_n to o_n. Due to Property 3.3 and Property 3.6, these parallel paths can be treated independently. These paths are simple, since each feedback connection (o_n, i_n) is considered only once in the algorithm. The verification algorithm only checks a subset of simple cyclic paths. Therefore, assuming $\mathbf{1} \prec g_c$ for every simple cyclic path c, the execution of the algorithm will terminate at step 1, declaring that the actor network is live.

We next prove the forward implication. Note that every output port o_j is only connected with one downstream port, i.e. i_j. Thus, without loss of generality, we can consider a cyclic path $c = (i_1, o_n, i_n, ...o_2, i_2, o_1, i_1)$, for some $n \in \mathbb{N}$, and $1 \leq n \leq N$. (For a cyclic path c' that begins with an output port, we can always consider its shifted version c that begins with an input port, since due to Corollary 3.16, $\mathbf{1} \prec g_{c'} \Leftrightarrow \mathbf{1} \prec g_c$.) We execute the verification algorithm by eliminating feedback connections in the following order: (o_n, i_n), (o_{n-1}, i_{n-1}), ..., (o_2, i_2). Eventually, we get $\delta_a(i_1, o_1)$. Assuming the network is live, $\mathbf{1} \prec \delta_a(i_1, o_1)$. Due to Lemma 3.17, $\delta_a(i_1, o_1) \leq g_c$. Thus we have $\mathbf{1} \prec g_c$.

In conclusion, the network is live if and only if for every simple cyclic path c in the dependency graph, $\mathbf{1} \prec g_c$. $\square$

In summary, there are two approaches for liveness analysis where the tag set $\mathcal{T}$ is totally ordered. The first approach is to use the verification algorithm described in Section 3.4.2. This approach requires a sequential procedure to treat one feedback connection at a time. The second approach is based on Theorem 3.18, which gives a necessary and sufficient condition for liveness. Due to Corollary 3.16, it is necessary and sufficient to check one cyclic path for each simple cycle. Using this approach, cyclic paths can be treated independently. Verification for each cycle can be performed in parallel. This approach is better at dealing with intersecting cycles.

When $\mathcal{T}$ is not totally ordered, the condition that "$\mathbf{1} \prec g_c$ for every simple cyclic path c" is neither a necessary condition for liveness (see Example 3.9), nor a sufficient

condition (see Example 3.7). Cycles cannot be considered independently. They must be resolved one at a time in a sequential manner.

Chapter 4

Applications

In this chapter, we study several models of computation and see how the causality interface theory can be applied to them.

4.1 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 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 3.18 presented in the previous chapter can be easily applied to all three models of computation.

4.1.1 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 [53].

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 3.5. From Corollary 3.14, we know that the causality interface of actor b in Figure 3.5(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 $T_0 \subset \mathcal{T}$. Since $\delta_b(q1, q2)$ is bounded by T_0, b is neither live nor causal. Thus,

Corollary 4.1 *Given a timed composite actor b as shown in Figure 3.5(b), actor b is causal if and only if actor a is causal and* $\mathbf{1} \prec \delta_a(p2, p3)$*, where* $\mathbf{1} = d_I$ *is the multiplicative identity.*

Following the verification algorithm described in Section 3.4.2, we can determine whether a network is causal. It follows naturally we have the following theorem about causality, a stronger property than liveness:

Theorem 4.2 *A finite network of continuous and causal timed actors is continuous and causal if and only if for every simple 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 4.1(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 simple cycles, 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 $\mathbf{1} \prec g_{c_2}$.

A second way to view this model is to create a hierarchy, as shown in Figure 4.1(b), and there is only one cycle between $q1$ and $q2$. The causality interface of actor b is given in (3.17), 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 Property 3.3). Therefore $\mathbf{1} \prec \delta_b(q1, q2) \Leftrightarrow$

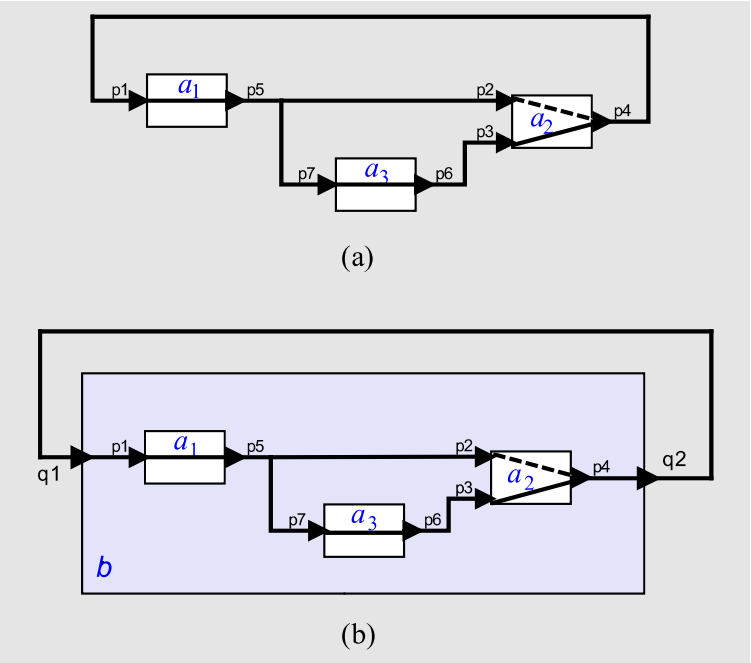


Figure 4.1: A timed model with a feedback loop. Figure (b) is an alternative way to treat the network by adding a layer of hierarchy.

$\mathbf{1} \prec g_{c_1}$ and $\mathbf{1} \prec g_{c_2}$ (due to Property 3.6). I.e., both approaches check for the same condition. This is an example that shows 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/causal.

In this example, we do not need to know exactly $\delta_{a_2}(p2, p4)$ but whether it is strict 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 strict causal interface in every cycle. In practice, most actors are causal or strict causal. Therefore, a simple technique to detect causality loop is to use the *boolean algebra* as the dependency algebra, where $D = \{true, false\}$ and $true < false$. The $\oplus$ operator is logic OR, and the $\otimes$ operator is logic AND. The additive identity $\mathbf{0}$ is $false$, and the multiplicative identity $\mathbf{1}$ is $true$. A causality interface $\delta_a(p, p') = false$ means that port p' has a strict causal dependency on p, and $\delta_a(p, p') = true$ means the dependency is just causal. A finite network of causal actors is live if and only if for every cyclic path c in the dependency graph, $true = \mathbf{1} < g_c$, or equivalently, $false = g_c$. However, note in theory, causality interfaces represented by the boolean algebra are not composable if there exists causality loop. A composition that involves causality loop is not causal, and thus its causality interfaces are neither $true$ nor $false$.

4.2 Application to Dataflow

In dataflow, the signals are streams of data tokens. Actors execute in response to the availability of data tokens. Although tags across different signals in dataflow only form a partial order, e.g., a tag t_1 at port p_1 may be incomparable to a tag t_2 at port p_2, tags in the same signal are totally ordered. Therefore, Theorem 3.18 can also be applied to dataflow models. Moreover, since the signals of dataflow are

sequences of tokens, the tags in a dataflow signal is order-isomorphic to $\mathbb{N}$. Thus we use $(\mathcal{D}(\mathbb{N}) \to \mathcal{D}(\mathbb{N}))$ as the dependency algebra. Further, 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 4.3 *A finite network of continuous and live dataflow actors is continuous and live if and only if for every simple cyclic path c in the dependency graph, $\mathbf{1} \prec g_c$, where $\mathbf{1} = d_I$ is the multiplicative identity.*

Wadge [68] 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.

4.2.1 Decidability

One question that might arise concerns decidability of deadlock. Theorem 4.3 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* [17], 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 4.3 implies that if for every simple 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 simple 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 [43], 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

$$\forall\ n \in \mathbb{N}_\infty, \quad (\delta_a(p_i, p_o))(n) = \begin{cases} \lfloor n/N \rfloor \cdot M + I, & \text{if } n < \infty \\ \infty, & \text{if } n = \infty, \end{cases} \tag{4.1}$$

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 4.4 *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

$$g = \bigoplus (\bigotimes \delta_a(p_i, p_o)), \tag{4.2}$$

where each $\delta_a(p_i, p_o)$ is in the form of (4.1), 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 (4.1), the following property holds:

$$\forall\ k, r \in \mathbb{N}, \quad \delta(kN + r) = \delta(r) + kM, \tag{4.3}$$

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

$$\forall\ k, r \in \mathbb{N}, \qquad \delta_a(kN_a + r) = \delta_a(r) + kM_a$$
$$\delta_b(kN_b + r) = \delta_b(r) + kM_b$$

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 (4.3). 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 (4.1), and $(\otimes\delta)$ satisfies (4.3). 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 $\mathbf{1} \prec \delta_2$. Due to Property 3.6,

$$\mathbf{1} \prec (\delta_1 \oplus \delta_2) \quad \Leftrightarrow \quad \mathbf{1} \prec \delta_1 \text{ and } \mathbf{1} \prec \delta_2.$$

Thus $\mathbf{1} \prec (\delta_1 \oplus \delta_2)$ is decidable. This generalizes easily to any expression of the form of (4.2) over a finite number of actors. □

Theorem 4.4 can be easily extended to cyclo-static data flow (CSDF) [14]. The proof is similar. The key is that a causality interface of a CSDF actor also satisfies (4.3) for some natural numbers N and M.

In [43], 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

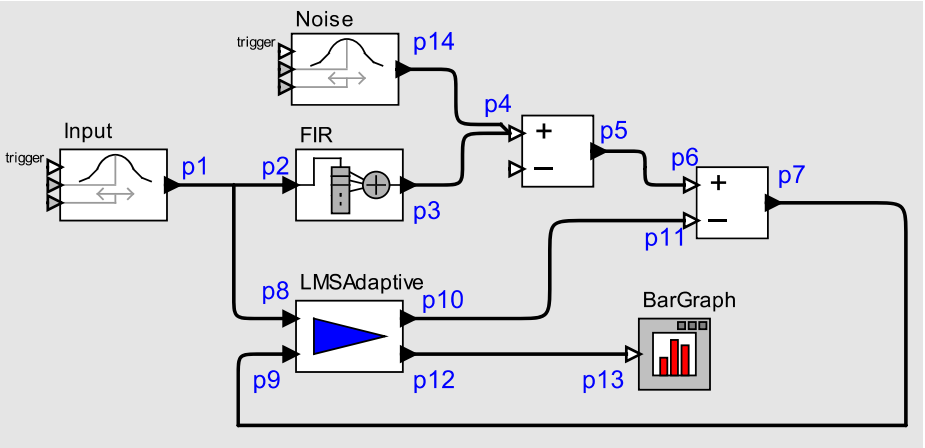


Figure 4.2: A dataflow example: least mean square adaptive filtering.

that provably terminates. The theory presented here 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 [43].

We now consider a dataflow example shown in Figure 4.2. This model shows that a least mean square (LMS) adaptive filter adapts its taps to a fixed FIR (finite impulse response) filter by observing the input and noisy output of the filter. All actors consume one token at their input ports and produce one token at their output ports on each firing of the corresponding actor. We observe that there is only one simple cycle in this model, i.e., $c = (p9, p10, p11, p7, p9)$. Therefore, we get:

$$\begin{aligned} g_c &= \delta(p9, p10) \otimes \delta(p11, p7) \\ &= d_I \otimes d_I \\ &= d_I \end{aligned}$$

Therefore this model is not live. The correct way is to increase the causality interfaces involved in the cycle so that $d_I \prec g_c$. It is not hard to see that we could add one initial token either at output port $p10$ or $p7$. Theorem 4.3 provides a criterion to allocate minimum correct number of initial tokens to avoid deadlock.

4.2.2 Relationship to Partial Metrics

Matthews uses a metric-space approach to treat deadlock [55]. 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 [55], this feedback system is deadlock-free if F_a is a contraction map in this complete partial

metric space, meaning

$$\exists\ c \in \mathbb{R}_0, \quad 0 \leq c < 1, \quad \text{such that}$$

$$\forall\ s_1, s_2 \in \mathcal{S}, \quad f(F_a(s_1), F_a(s_2)) \leq c f(s_1, s_2).$$

Theorem 4.5 *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 prefix of s_2. Therefore, we have[1],

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

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

[1]We define $2^{-\infty} = 0$.

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 $\mathbf{1} \prec \delta_a$. $\square$

Matthews further studied liveness conditions for dataflow actor networks with more than one cycle [55]. Consider an actor network with a set of ports P. A *cycle contraction constant* is a function $k : P^2 \rightarrow \mathbb{R}$ such that for every cyclic path $c = (p_1, p_2, ..., p_n, p_1)$ in the dependency graph,

$$k(p_1, p_2) \cdot k(p_2, p_3) \cdot ... \cdot k(p_n, p_1) < 1.$$

A function $F_a : \mathcal{S}^N \rightarrow \mathcal{S}^N$ with input ports P_i and output ports P_o is a *cycle contraction* if there exists a cycle contraction constant $k : (P_i \cup P_o)^2 \rightarrow \mathbb{R}$ such that

$$\forall\ s, s' \in \mathcal{S}^N,\ \forall\ p_i \in P_i,\ p_o \in P_o,$$

$$f(\pi_{\{p_o\}}(F_a(s)), \pi_{\{p_o\}}(F_a(s'))) \leq k(p_i, p_o) \cdot f(\pi_{\{p_i\}}(s), \pi_{\{p_i\}}(s')).$$

In Theorem 5.1 of [55], Matthews showed that a function $F_a : \mathcal{S}^N \rightarrow \mathcal{S}^N$ has a complete least fixed point if F_a is a cycle contraction. However, this is a sufficient but not a necessary condition for liveness of dataflow actor networks.

Theorem 4.6 *Let a be a continuous dataflow actor with N input ports and N output ports. The actor function is F_a. The i-th output port of a is connected back to the i-th input port. If F_a is a cycle contraction, then for every cyclic path c in the dependency graph, $\mathbf{1} \prec g_c$.*

This statement is consistent with my results.

PROOF. Let $k : (P_i \cup P_o)^2 \to \mathbb{R}$ be a cycle contraction constant of F_a, where P_i is the set of input ports and P_o the set of output ports of a. $\forall\ p_i \in P_i$ and $p_o \in P_o$, consider $s, s' \in \mathcal{S}^N$ such that $\pi_{\{p_i\}}(s)$ is of finite length n, $\pi_{\{p_i\}}(s) \sqsubseteq \pi_{\{p_i\}}(s')$, and

$$\forall\ p \in P_i \text{ and } p \neq p_i,$$

$$\pi_{\{p\}}(s) = \pi_{\{p\}}(s') \text{ and } |\pi_{\{p\}}(s)| = |\pi_{\{p\}}(s')| = \infty.$$

I.e., s and s' are identical and complete at all input ports but p_i. Therefore, we have

$$f(\pi_{\{p_i\}}(s), \pi_{\{p_i\}}(s)) = 2^{-n},$$

$$f(\pi_{\{p_o\}}(F_a(s)), \pi_{\{p_o\}}(F_a(s'))) = 2^{-\delta_a(p_i,p_o)(n)}.$$

Since k is a cycle contraction constant of F_a, we have

$$2^{-\delta_a(p_i,p_o)(n)} \leq k(p_i, p_o) \cdot 2^{-n}.$$

For a connector c that connects an output port p_o of a back to an input port p_i, $1 \leq k(p_o, p_i)$. This is because

$$2^{-\delta_c(p_o,p_i)} = 2^{-n} \leq k(p_o, p_i) \cdot 2^{-n}.$$

Now consider any cyclic path $c = (p_1, p_2, ..., p_n)$ in the dependency graph. Therefore, we have

$$2^{-\delta(p_1,p_2)(n)} \leq k(p_1, p_2) \cdot 2^{-n},$$

$$2^{-\delta(p_2,p_3)(\delta(p_1,p_2)(n))} \leq k(p_2, p_3) \cdot 2^{-\delta(p_1,p_2)(n)} \leq k(p_1, p_2) \cdot k(p_2, p_3) \cdot 2^{-n},$$

$$\vdots$$

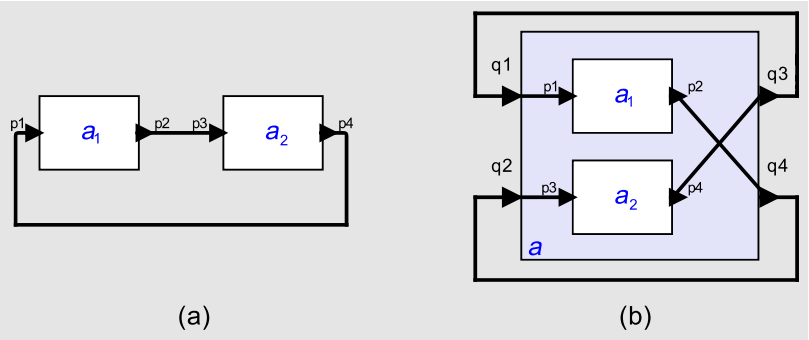


Figure 4.3: A dataflow example that is live yet not a cycle contraction.

Eventually, we get

$$2^{-(\delta(p_1,p_2)\otimes\ldots\otimes\delta(p_n,p_1))(n)} \leq k(p_1,p_2)\cdot\ldots\cdot k(p_n,p_1)\cdot 2^{-n} < 2^{-n}.$$

I.e.,

$$2^{-g_c(n)} < 2^{-n}.$$

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

An example is sufficient to prove that cycle contraction is not a necessary condition for liveness. Consider the example in Figure 4.3. Figure 4.3(b) is the same model as Figure 4.3(a) but interpreted as a composite actor a with two feedback loops. The causality interfaces of a_1 and a_2 are

$$\forall\ n \in \mathbb{N}, \quad \delta_{a_1}(n) = \lfloor n/2 \rfloor, \quad \delta_{a_2}(n) = 2n+2, \text{ and,}$$

$$\delta_{a_1}(\infty) = \delta_{a_2}(\infty) = \infty,$$

where we have dropped the notation of ports in the δ's, since there is only one interface for each actor a_1 and a_2.

It is easy to verify by using Theorem 4.3 that the model in Figure 4.3 is live. (The condition that $\mathbf{1} \prec g_c$ for every cyclic path c holds, no matter whether we view the network as in Figure 4.3(a) or in 4.3(b).) However, the composite actor a in Figure 4.3(b) is not a cycle contraction.

PROOF. Consider two input signals $s, s' \in \mathcal{S}^2$, where $\pi_{\{q_1\}}(s)$ is of finite length n and a prefix of $\pi_{\{q_1\}}(s')$. Therefore,

$$f(\pi_{\{q_1\}}(s), \pi_{\{q_1\}}(s')) = 2^{-n},$$
$$f(\pi_{\{q_4\}}(F_a(s)), \pi_{\{q_4\}}(F_a(s'))) = 2^{-\lfloor n/2 \rfloor}.$$

If there exists a $k \in \mathbb{R}$ such that

$$f(\pi_{\{q_4\}}(F_a(s)), \pi_{\{q_4\}}(F_a(s'))) \leq k \cdot f(\pi_{\{q_1\}}(s), \pi_{\{q_1\}}(s')),$$

then $\forall\ n \in \mathbb{N}$,

$$2^{-\lfloor n/2 \rfloor} \leq k \cdot 2^{-n}.$$

I.e.,

$$n - \lfloor n/2 \rfloor \leq \log_2(k).$$

This contradicts the fact that $n - \lfloor n/2 \rfloor$ does not have an upper bound. Therefore, actor a in Figure 4.3(b) is not a cycle contraction. □

From the above example, we see that Matthews' theorem rules out many actor networks that are in fact live. My theorem is, by contrast, tight. Moreover, it may be hard to determine whether a function F_a is a cycle contraction. In particular, finding

the cycle contraction constant $k : P^2 \to \mathbb{R}$ for F_a is hard, not to mention that it requires to consider intersecting cycles collectively. My theorem provides a criterion that is much easier to verify, and I showed that intersecting cycles can be treated independently and in parallel.

Chapter 5

Ordering Dependencies: The General Story

In this chapter, I study the causality properties of an actor in the general sense, i.e., actors that are not abstracted by functions. I present a mathematical structure called *ordering dependency*. Ordering dependency is also an interface model that captures the causality properties of actors. It is closest to the spirit of the tagged signal model. I show how ordering dependencies can be used to analyze rendezvous of sequential programs, and used in scheduling.

5.1 Ordering Constraints

The essence of causality interfaces for functional actors is to reflect the ordering constraints on output events imposed by input events. I.e., certain output events

cannot occur before certain input events have occurred. One may wonder whether there exists interface model for an actor in the general sense, and how far we can push the theory. Recall that an actor a with N ports is a set of behaviors. The actor asserts constraints on the signals at its ports. The causality property of a is therefore the projection of its behaviors onto the tag set. I.e., $\forall\ s \in a$, $\text{dom}(s)$ is a possible ordering of the events in the signals of a. Therefore, the ordering constraints of events of a is a set

$$\text{dom}(a) = \{\text{dom}(s) \mid s \in a\}. \tag{5.1}$$

$\text{dom}(a)$ is a subset of $(\mathcal{D}(\mathcal{T}))^N$.

Whether such ordering constraints should be represented as constraints on pairs of ports depends on two questions. The first question is whether we *can* do it, i.e., whether the ordering constraints projected onto pairs of ports reflect, or at least conservatively approximate the ordering constraints on the actor. Recall that a subset of $(\mathcal{D}(\mathcal{T}))^N$ can be viewed as a relation on $(\mathcal{D}(\mathcal{T}))^N$. Hence the question becomes whether such relation is reducible to a set of binary relations. In general, relation reducibility to lower dimensions is a difficult problem, even for a simple tag set $\mathcal{T}$ and small N. But before we dig ourselves into mathematic theorems, let us first assume we are able to classify actors with ordering constraints reducible to binary relations. Hence the second question is whether we *want* to do it. Will such projection to binary relations make our analysis easier? To answer this question, we summarize the major features of the problem in this chapter as compared to that in the previous chapters.

1. **More edges in the dependency graph.**

   Ordering constraints for functional actors only take place on an output port imposed by an input port. While actors in the general sense, could have ordering constraints between any pair of ports. Even if we have a vague notion of "input" and "output" ports by differentiating the direction of the signal flows, ordering constraints could still be between a pair of input ports, as well as between a pair of output ports. (We will see such examples when we discuss the rendezvous semantics in Section 5.2.) Therefore, under the functional semantics, an actor with N input ports and N output ports will have N^2 edges in the dependency graph. An actor with $2N$ ports, in the general sense, will have $2N(2N-1) \approx 4N^2$ edges. Therefore, a dependency graph for an actor network in the general sense has more edges.

2. **Edges in the dependency graph are bi-directional.**

   Functional actors have deterministic behaviors. This means that what happens downstream (in the sense of signal flow) cannot affect the behaviors upstream. However, ordering constraints are mutual. Compositions downstream may impose constraints upstream so that certain behaviors are not possible. (Again, we will see examples in Section 5.2.) If we use an edge to represent the existence of ordering constraints between two ports, then such a dependency graph is bi-directional. This significantly increases the connectivity of ports and the

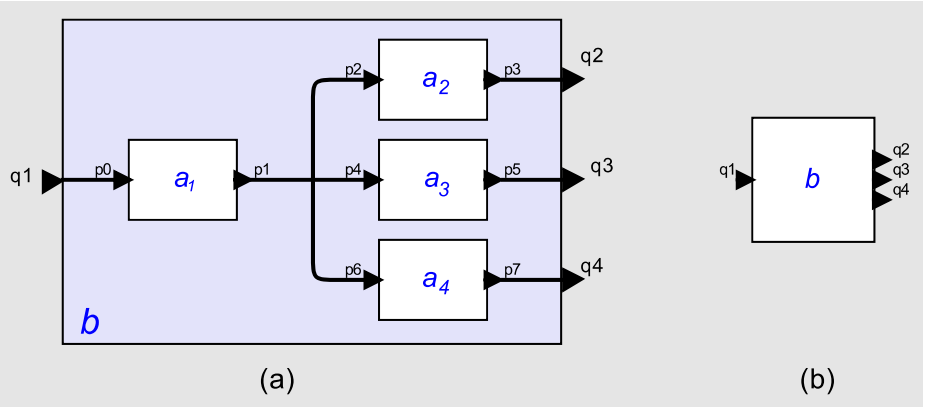


Figure 5.1: A composition of four actors.

number of cycles. For example, consider the composition in Figure 5.1. If all actors are functional, the dependency graph (in the sense of causality interfaces) of the composite actor b is shown in Figure 5.2(a). While in the general case, the dependency graph is as shown in 5.2(b). If we represent ordering constraints on actor b based on pairs of ports, we need to represent such constraints for six pairs of ports. Moreover, one can prove that Figure 5.2(b) contains seven simple cycles[1]. Thus projecting ordering constraints onto pairs of ports does not make our analysis any easier. It is more straightforward to address the ordering constraints on actors collectively. I.e., ordering constraints should be defined based on actors, not on pairs of ports. This is not surprising. Since ordering constraints could exist between any pair of ports, there is little we can orthogonalize our concerns on.

[1]Since Figure 5.2(b) is a bi-directional graph, here we consider two cyclic paths with reversing order of ports refer to the same cycle.

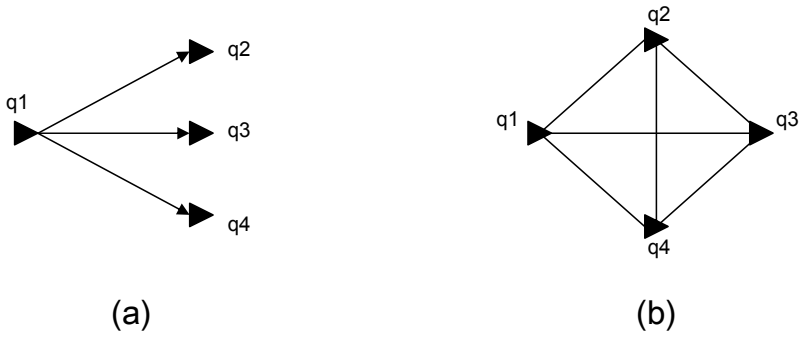


Figure 5.2: Differences between dependency graphs of causality interfaces and ordering dependencies. Figure (a) shows the dependency graph for causality interfaces, which is a directed graph, for the composition in Figure 5.1. Figure (b) shows the dependency graph for ordering dependencies of the same composition. The dependency graph for ordering dependencies is a bi-directional graph with more edges and intersecting cycles.

In order to differ from the causality interfaces presented in Chapter 3, I will address the following mathematical structure as *ordering dependency*. However, it should be understood that causality interface and ordering dependency are both interface models that capture causality properties of an actor. With certain assumptions, the former is more amenable for analysis, while the latter is more general.

An ordering dependency Φ_a of an actor a is a set of functions.

$$\Phi_a = (P_a \rightarrow \mathcal{D}(\mathcal{T})), \tag{5.2}$$

where P_a is the set of ports of a. $\forall\ \phi \in \Phi_a$, ϕ is a function that maps a port of a to a down set of $\mathcal{T}$. P_a is also called the *domain* of ϕ. It is the set of ports where ϕ is defined, written as $\mathrm{dom}(\phi)$. The *range* of ϕ is a down set:

$$\mathrm{range}(\phi) = \bigcup\{\phi(p) \mid p \in \mathrm{dom}(\phi)\}.$$

$\forall\ t \in \mathrm{range}(\phi)$, the *preimage* of t is a set of ports such that $\forall\ p \in \mathrm{preimage}(t)$,

$t \in \phi(p)$.

Take an example of an actor a with one input port $p1$ and one output port $p2$ in a Kahn-MacQueen process network (KPN). The actor produces one output event for each input event. In the KPN model of computation, the tag set of a signal s is *order-isomorphic* to a subset of the natural numbers. Two posets P and Q are said to be order-isomorphic if there exists a function $f : P \rightarrow Q$ such that $\forall\ x, y \in P$, $x \leq_P y$ if and only if $f(x) \leq_Q f(y)$.

$\forall\ T \in \mathcal{D}(\mathcal{T})$ of the Kahn-MacQueen process network, we use $T(i)$ to denote the tag of the i-th event in T, for some positive natural number i. Thus, it should be understood that $\forall\ i, j \in \mathbb{N}_+$, $i < j\ \Rightarrow\ T(i) < T(j)$, where $\mathbb{N}_+ = \{1, 2, ...\}$ is the set of positive natural numbers.

The ordering dependency of a is therefore given by:

$$\Phi_a = \{\phi \mid \forall\ i \in \mathbb{N}_+,\ \ (\phi(p2))(i) \text{ exists} \\ \Rightarrow\ (\phi(p1))(i) \text{ exists, and } (\phi(p1))(i) < (\phi(p2))(i)\}. \quad (5.3)$$

Intuitively, (5.3) says that $\forall\ i \in \mathbb{N}_+$, the i-th event at port $p1$ happens before the i-th event at port $p2$.

The ordering constraints on a connector c that connects port P_c simply asserts that the tag sets of signals at each port $p \in P_c$ are identical. Therefore, the ordering dependency for c is:

$$\Phi_c = \{\phi \mid \exists\ T \in \mathcal{D}(\mathcal{T}) \text{ such that } \forall\ p \in P_c,\ \phi(p) = T\}. \quad (5.4)$$

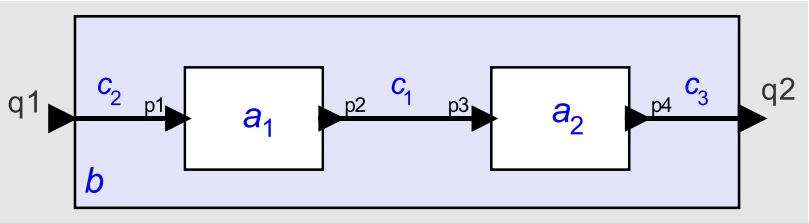


Figure 5.3: A composition of actors in process network.

Since an ordering dependency represents the ordering constraints on events of an actor, a composition of the ordering dependencies is the intersection of these constraints. Given an actor a with ordering dependency Φ_a and an actor b with ordering dependency Φ_b, the ordering dependency of the composition of a and b is a set of functions $\Phi_{a,b} = ((P_a \cup P_b) \rightarrow \mathcal{D}(\mathcal{T}))$, and,

$$\Phi_{a,b} = \Phi_a \wedge \Phi_b = \{\phi \mid \phi \downarrow \mathrm{dom}(\Phi_a) \in \Phi_a \text{ and } \phi \downarrow \mathrm{dom}(\Phi_b) \in \Phi_b\},$$

where $\phi \downarrow \mathrm{dom}(\Phi_a)$ means that the function ϕ is restricted to the domain of Φ_a. The notation of $\phi \downarrow \mathrm{dom}(\Phi_b)$ is similar.

We look at an example of composition shown in Figure 5.3. Actor b is composed of actor a_1, a_2, and connector c_1, c_2 and c_3. Thus their intersection is

$$\Phi = \Phi_{a_1} \wedge \Phi_{a_2} \wedge \Phi_{c_1} \wedge \Phi_{c_2} \wedge \Phi_{c_3}. \tag{5.5}$$

Note that the ordering of the actors and connectors does not matter, since the $\wedge$ operation is commutative. However, (5.5) is *yet not* the ordering dependency of b, since $\forall\ \phi \in \Phi$, $\mathrm{dom}(\phi)$ involves ports other than the ports of b. Φ_b is the projection

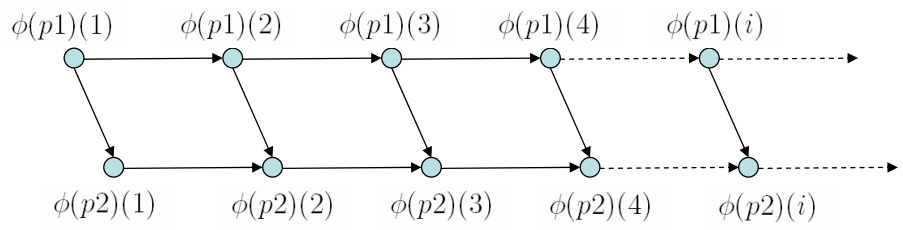


Figure 5.4: Diagram of ordering dependency of an actor in Kahn Process Network.

of Φ onto the (external) ports of b, written as,

$$\Phi_b = \Phi \downarrow \{q1, q2\}.$$

5.1.1 Ordering Dependency as Set of Posets

There are two benefits of choosing (5.2) as the formulation of ordering dependency, rather than defining it as a set to tags, as in (5.1). First, it is easy to address the port at which there is an event of tag t. Secondly, the range of each $\phi \in \Phi_a$ is a down set of $\mathcal{T}$. Therefore, range(ϕ) is a poset induced from $\mathcal{T}$. Thus, we can address the partial order between tags across two ports.

We again use the example of a KPN actor a with one input port $p1$ and one output port $p2$. The ordering dependency of a is specified in (5.3). $\forall\ \phi \in \Phi_a$, (5.3) specifies a partial order on $\phi(p1) \cup \phi(p2)$. It might be more explicit to the readers when we draw the diagram of such partial order, which is shown in Figure 5.4. Any ϕ that satisfies such diagram in Figure 5.4 is an element in Φ_a.

To form the diagram for the partial order of a connector, we first merge nodes

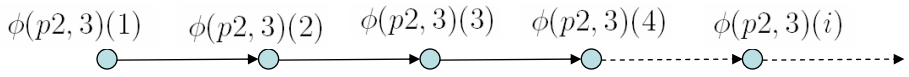


Figure 5.5: Diagram of ordering dependency of a connector.

with the same tag. Figure 5.5 shows such diagram for a connector that connects port $p2$ and $p3$. It should be understood that node $(\phi(p2,3))(i)$ represents $(\phi(p2))(i)$ as well as $(\phi(p3))(i)$.

In order to address the composition of ordering dependencies from a partial-order perspective, we first need to define some terms. $\forall\ \phi_1 \in \Phi_a$, $\phi_2 \in \Phi_b$, ϕ_1 and ϕ_2 is said to be *unionable* if

1. $\forall\ p \in \text{dom}(\phi_1) \cap \text{dom}(\phi_2)$, $\phi_1(p) = \phi_2(p)$.

2. range(ϕ_1) and range(ϕ_1) are orderly unionable.

If ϕ_1 and ϕ_2 are unionable, we define the *union* of ϕ_1 and ϕ_2 to be a function $(\phi_1 \tilde{\cup} \phi_2) : (\text{dom}(\phi_1) \cup \text{dom}(\phi_2)) \rightarrow \mathcal{D}(\mathcal{T})$. I.e.,

$$(\phi_1 \tilde{\cup} \phi_2)(p) = \begin{cases} \phi_1(p), & \text{if } p \in \text{dom}(\phi_1) \\ \phi_2(p), & \text{otherwise,} \end{cases}$$

where the range of $\phi_1 \tilde{\cup} \phi_2$ is a poset that inherits the partial order of (range(ϕ_1)) $\tilde{\cup}$ (range(ϕ_2)).

In other words, the ordering dependency of the composition of a and b is a set of possible unions of ϕ_1 and ϕ_2, where $\phi_1 \in \Phi_a$ and $\phi_2 \in \Phi_b$. I.e.,

$$\Phi_{a,b} = \{\phi_1 \tilde{\cup} \phi_2 \mid \phi_1 \in \Phi_a, \phi_2 \in \Phi_b, \text{ and } \phi_1 \tilde{\cup} \phi_2 \text{ exists.}\}$$

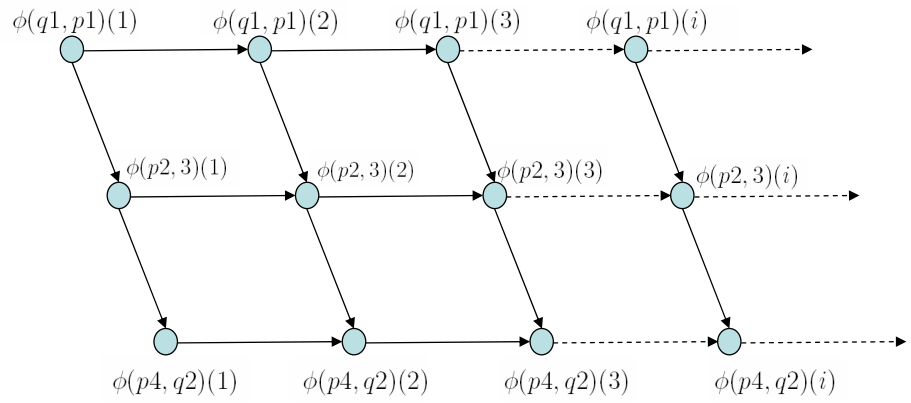


Figure 5.6: Diagram of the partial order representing composition of ordering dependencies for Figure 5.3.

Since the composition of ordering dependencies corresponds to a set of possible order unions of posets, we could use diagram composition to find the composite ordering dependency. We again use the example shown in Figure 5.3. We assume the ordering dependencies of actors a_1 and a_2 are given in (5.3). The diagram composition of a_1, a_2 with connectors c_1, c_2 and c_3 is shown in Figure 5.6. Similar to previous analysis, a projection onto external ports $q1$ and $q2$ is necessary before we get the diagram of the ordering dependency of b. This represents the partial order on the tags at $q1$ and $q2$ induced from the tags at ports $\{q1, p1, p2, p3, p4, q2\}$. The projected diagram is shown in Figure 5.7.

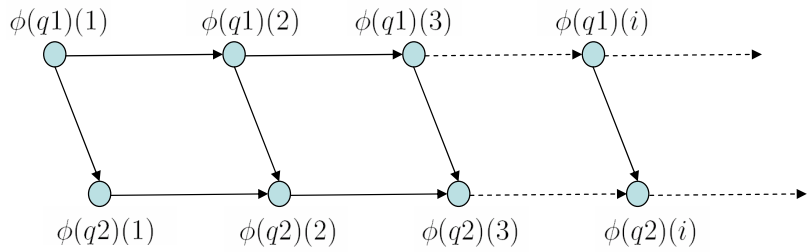


Figure 5.7: Diagram of the composed ordering dependency for actor b in Figure 5.3. This diagram represents the partial order on tags of external ports $q1$ and $q2$, induced from the partial order represented by Figure 5.6.

5.2 Use of Ordering Dependencies in Rendezvous of Sequential Programs

A *rendezvous* constrains sequential programs reaching a particular point to verify that another program has reached a corresponding point before proceeding. Representatives of models of computation that involve rendezvous include the communicating sequential processes (CSP) of Hoare [37] and calculus of communicating systems (CCS) of Milner [56].

Following the tagged signal model concept, rendezvous is a constraint on the order of events such that certain events must have the same tag [49]. Such constraints cannot be captured by a functional actor. Therefore, we cannot use the causality interface framework presented in Chapter 3. Instead, we need to use ordering dependencies.

5.2.1 Examples of Sequential Programs

Consider a simple actor a with one input port $p1$ and one output port $p2$. The sequential program that implements a is given in Figure 5.8.

Recall that signals of a sequential program are sequences of data tokens. For some down set $T \in \mathcal{D}(\mathcal{T})$ where such a signal is defined, we use $T(i)$ to denote the tag of the i-th event in T for some positive natural number i. Therefore, the ordering dependency of a is given by:

$$\begin{aligned}\Phi_a = \{(\phi \mid \ & \forall\, i \in \mathbb{N}_+, \\ & \phi(p2)(i) \text{ exists } \Rightarrow \phi(p1)(i) \text{ exists, and } \phi(p1)(i) < \phi(p2)(i), \\ & \phi(p1)(i+1) \text{ exists } \Rightarrow \phi(p2)(i) \text{ exists, and } \phi(p2)(i) < \phi(p1)(i+1)\}.\end{aligned} \tag{5.6}$$

Intuitively, this says that the i-th input event of a happens before the i-th output event, which happens before the $(i+1)$-th input event. Obviously, (5.6) is a subset of (5.3). This is not surprising since the sequential program in Figure 5.8 only implements a subset of behaviors of the process network actor. Figure 5.9 shows the diagram of the ordering dependency of this sequential program.

We now consider and compare two other sequential programs, given in Figure 5.10. Actor a_1 consists of one input port and two output ports. Actor a_2 consists of two input ports and one output port. It is not difficult to see that their ordering dependencies are both given by:

```
Actor a (In p1, Out p2) {
    repeat {
        t = get(p1);
        send(p2, t);
    }
}
```

Figure 5.8: A sequential program.

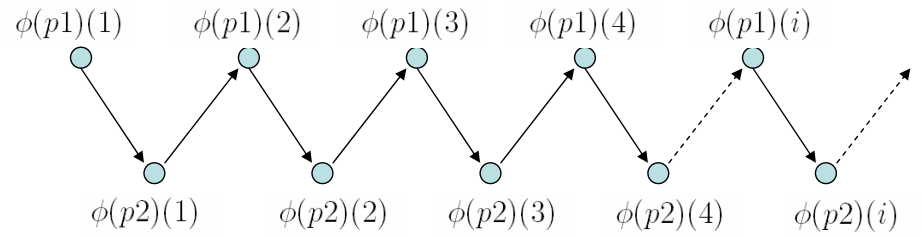


Figure 5.9: Diagram of the ordering dependency of actor a in Figure 5.8.

$$\begin{aligned}\Phi = \{(\phi \mid \ & \forall\, i \in \mathbb{N}_+, \\ & \phi(p2)(i) \text{ exists } \Rightarrow \phi(p1)(i) \text{ exists, and } \phi(p1)(i) < \phi(p2)(i), \\ & \phi(p3)(i) \text{ exists } \Rightarrow \phi(p2)(i) \text{ exists, and } \phi(p2)(i) < \phi(p3)(i), \\ & \phi(p1)(i+1) \text{ exists } \Rightarrow \phi(p3)(i) \text{ exists, and } \phi(p3)(i) < \phi(p1)(i+1)\}.\end{aligned} \tag{5.7}$$

The diagram that represents the partial order specified by (5.7) is given by Figure 5.11.

This is an example where two actors appear to have identical ordering dependency (which is an interface that captures the causality properties). The only difference is that port $p2$ is an output port in the first program, but an input port in the second

```
Actor a1 (In p1, Out p2, p3) {
    repeat {
        t = get(p1);
        send(p2, t);
        send(p3, t);
    }
}
```

(a)

```
Actor a2 (In p1, p2, Out p3) {
    repeat {
        t1 = get(p1);
        t2 = get(p2);
        send(p3, t1 + t2);
    }
}
```

(b)

Figure 5.10: Two sequential programs that have the same ordering dependency.

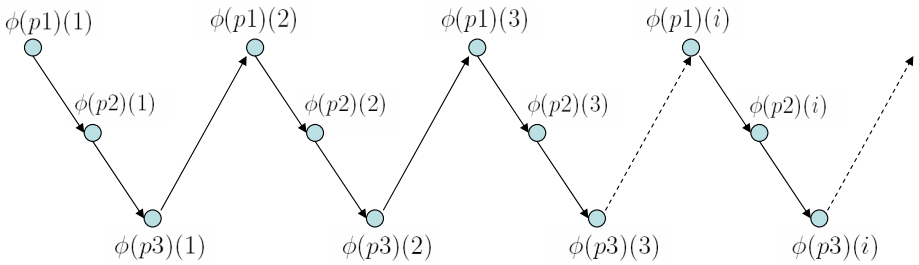


Figure 5.11: Diagram of the ordering dependency for both actor a_1 and actor a_2 in Figure 5.10.

program. Can we treat these two actors as identical in causality analysis? The answer depends on the semantics of the network. If the semantics is described by the ordering of events (but does not distinguish whether the event is an input or an output one), then we can treat these two actors as identical. The rendezvous semantics that we will be discussing below falls into this category. If the semantics does distinguish input and output events, then we cannot treat the two actors identically. In Section 5.3, I will discuss the use of ordering dependencies in scheduling. The schedule to be found describes an execution order of input events. It is not hard to imagine that input and output ports are treated differently.

5.2.2 Rendezvous

A rendezvous in actor networks may be formed in two ways:

1. *Rendezvous through multi-way communication.*

   Consider the example in Figure 5.12. Note that port $p2$ and $p4$ are connected to the same output $p1$. Due to the ordering dependencies imposed by connector c_1 and c_2, respectively, we know that signals at the two input ports $p2$ and $p4$ must rendezvous, i.e., they must have the same tags. If there exist other constraints on $p2$ and $p4$ that are in conflict with this requirement, then the rendezvous cannot be formed. This is an example where ordering constraints could be among a set of input ports.

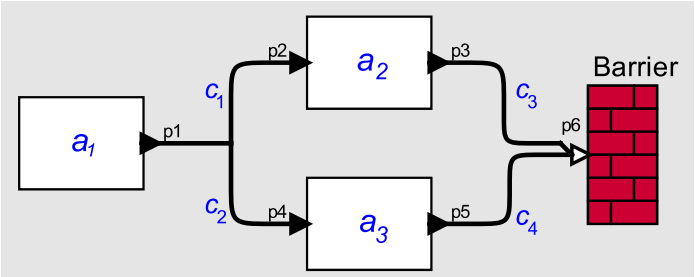


Figure 5.12: A rendezvous example.

2. *Rendezvous through the barrier actor.*

   Secondly, a rendezvous can also be formed using a *barrier* actor. All signals connected to the same barrier actor must rendezvous with each other. The ordering dependency of a barrier actor is

$$\Phi_{barrier} = \{\phi \mid \exists\ T \in \mathcal{D}(\mathcal{T}) \text{ such that } \forall\ p \in P_{barrier},\ \phi(p) = T\},$$

   where $P_{barrier}$ is the set of ports of the barrier actor. In Figure 5.12, connector c_3 and c_4 are both connected to a barrier actor. This constrains that signals at port $p3$ and $p5$ must have identical tags. This is an example where ordering constraints could exist among a set of output ports.

From the above examples, we see that a rendezvous may impose dependency among arbitrary set of ports. The ordering dependency captures the causality properties of an actor collectively, and therefore can serve this purpose.

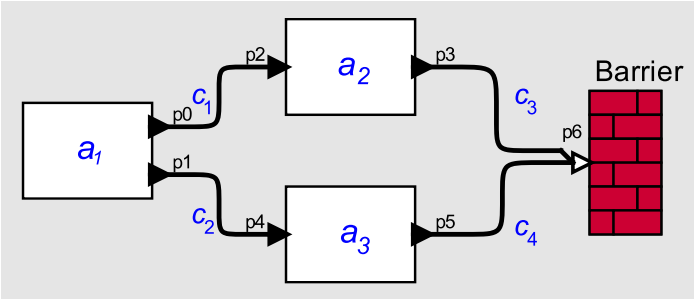


Figure 5.13: A more complicated rendezvous example.

```
Actor a1 (Out p0, p1) {
  repeat {
    send(p0, 1);
    send(p1, 1);
  }
}
```

Figure 5.14: The sequential program that implements actor a_1 in Figure 5.13.

Given a set A of actors and a set C of connectors, the ordering dependency of the composition is

$$\Phi = \bigwedge_{i \in A \cup C} \Phi_i.$$

Since the actors are sequential, the signal at a particular port p is a sequence of events. A rendezvous that involves the n-th event at port p cannot be formed if there exists no $\phi \in \Phi$ such that $\phi(p)(n)$ exists. For example, we consider the model in Figure 5.13. The ordering dependency of the composition is

$$\Phi = \Phi_{a_1} \wedge \Phi_{a_2} \wedge \Phi_{a_3} \wedge \Phi_{c_1} \wedge \Phi_{c_2} \wedge \Phi_{c_3} \wedge \Phi_{c_4} \wedge \Phi_{barrier}. \tag{5.8}$$

We assume that actor a_1 is implemented by the sequential program shown in Figure 5.14. Therefore, the ordering dependency of a_1 is

$$\begin{aligned}\Phi_{a_1} = \{(\phi \mid \ & \forall\, i \in \mathbb{N}_+, \\ & \phi(p1)(i) \text{ exists } \Rightarrow \phi(p0)(i) \text{ exists, and } \phi(p0)(i) < \phi(p1)(i), \\ & \phi(p0)(i+1) \text{ exists } \Rightarrow \phi(p1)(i) \text{ exists, and } \phi(p1)(i) < \phi(p0)(i+1)\}.\end{aligned}$$

We discuss three cases of actors a_2 and a_3.

1. Actors a_2 and a_3 are implemented by the program shown in Figure 5.8.

2. Actor a_2 is a connector. Actor a_3 is implemented by the sequential program shown in Figure 5.8.

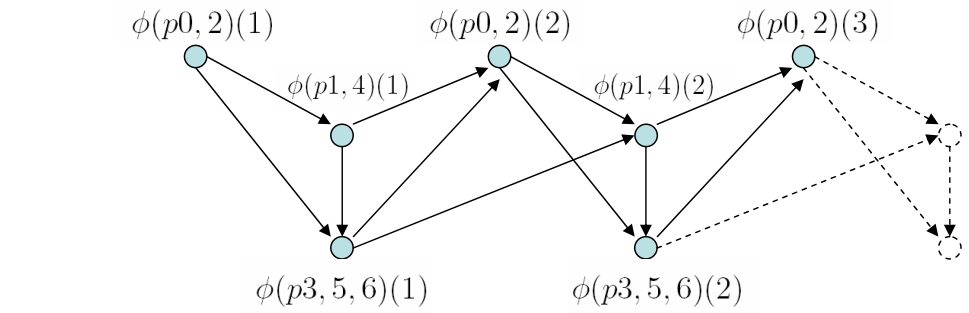
3. Actor a_3 is a connector. Actor a_2 is implemented by the sequential program shown in Figure 5.8.

In the first case, the ordering dependencies of a_2 and a_3 are given by (5.6). The ordering dependencies of a_1, the connectors and the barrier actor are given from

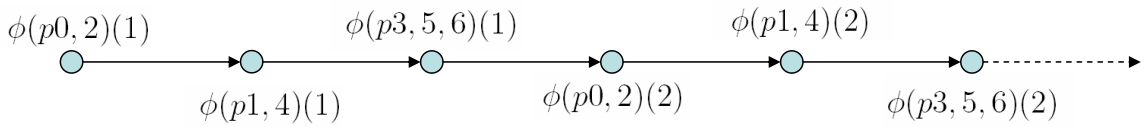
above. By applying the ordering dependencies to (5.8), we eventually get

$$\begin{aligned}\Phi = \{\phi \mid \quad & \phi(p0) = \phi(p2), \\ & \phi(p1) = \phi(p4), \\ & \phi(p3) = \phi(p6) = \phi(p5); \\ & \forall\ i \in \mathbb{N}_+, \\ & \phi(p1)(i) \text{ exists } \Rightarrow \phi(p0)(i) \text{ exists, and } \phi(p0)(i) < \phi(p1)(i), \\ & \phi(p3)(i) \text{ exists } \Rightarrow \phi(p1)(i) \text{ exists, and } \phi(p1)(i) < \phi(p3)(i), \\ & \phi(p0)(i+1) \text{ exists } \Rightarrow \phi(p3)(i) \text{ exists, and } \phi(p3)(i) < \phi(p0)(i)\}.\end{aligned}$$

Recall that an ordering dependency is a set of posets. If the structure of a sequential program is sufficiently regular, its ordering dependency may be represented diagrammatically. It is easier to reason about composition using graph composition. A rendezvous cannot be formed if the composition causes the corresponding tag (node) involved in a loop. Such tag must be eliminated in the graph. Otherwise, it violates the partial order on the tag set $\mathcal{T}$. Figure 5.15(a) shows the composition procedure of the example in Figure 5.13. Note that it is not yet a diagram since it includes edges that are not covering relations. Figure 5.15(b) is the diagram obtained by transitive reduction of Figure 5.15(a). It explicitly shows the order of the different rendezvous actions: a rendezvous between ports $p0$ and $p2$ is first formed, then between ports $p1$ and $p4$, and then among ports $p3$, $p6$ and $p5$. This order is then repeated. This diagram can be arbitrarily long, suggesting that a rendezvous can always be formed.



(a)



(b)

Figure 5.15: Diagrammatical analysis of rendezvous in the model in Figure 5.13. Figure (a) shows graph composition of the component ordering dependencies. However, note that it is not a diagram, since it includes edges that are not covering relations. Figure (b) is the diagram obtained from Figure (a) by transitive reduction.

For the second case, where a_2 is replaced with a connector, the analysis is similar. In particular, it is easy to find using the graphical approach that a rendezvous can always be formed. However, for the third case, where a_3 is a connector and a_2 is implemented by the program in Figure 5.8, a rendezvous will never be formed.

At this point, a question might arise: why didn't we care about causality properties between a pair of input ports or a pair of output ports in the previous chapters? A timed or dataflow actor may also be implemented using sequential programs. Is our formalization of causality interfaces for functional actors wrong? The key to the answer is that functional actors and the semantics of composition (e.g., the least fixed point semantics) are fully captured by their functions, which are mappings from a set of input ports to a set of output ports. Therefore, projections of behavioral constraints onto pairs of input and output port are sufficient. Think: whether the actors are implemented by sequential programs does not change the least fixed point of the actor network. Therefore, we could simply assume that there are no ordering dependencies between a pair of input ports, or a pair of output ports.

As regards to this section, rendezvous semantics is not expressible by functions. Therefore, causality properties that are not captured in the actor functions must be considered for those actors that rendezvous with each other (even if the actors themselves are functional).

5.3 Use of Ordering Dependencies in Scheduling

In [74], Zhao et al. presented a PTIDES (Programming Temporally Integrated Distributed Embedded Systems) model. The PTIDES model extended my work on causality interface theory to find a schedule for time-synchronized distributed real-time systems. Zhao et al. consider input ports that affect the same output port to be *equivalent*. The set of equivalent ports forms a *equivalence class*. A *relevant dependency* is a function

$$\delta : Q \times Q \rightarrow D, \tag{5.9}$$

where Q is the set of equivalence classes in an actor network, and D is a dependency algebra.

A *relevant dependency graph* is constructed by merging nodes (ports) that are equivalent in the dependency graph. The PTIDES model can be viewed as a specific case that sits between the formulation of causality interface and that of ordering dependency. In [74], the actor networks of interest are composed of functional actors and use the least fixed point semantics. Therefore, ordering dependency between an input and an output port could be captured by the causality interface model presented in Chapter 3. However, Zhao et al. are not only interested in determining whether the least fixed point is complete. They are also interested in finding an execution order of input events to find that least fixed point. I.e., ordering dependencies among input ports become important. The equivalence class says an actor shall not be scheduled for execution until all of its input ports are ready. A relevant dependency graph views

input ports collectively based on actors (or equivalently, it adds bi-directional edges among input ports of the same actor), while edges between input and output pairs remain directed.

As a summary, I have presented two mathematical structures. The causality interface is useful when an actor network is composed of functional actors and the semantics is solely determined by the actor functions. The ordering dependency is a more general formulation, which captures causality properties that are not expressible by functions. Similar to the tagged signal meta model, the notation of ordering dependencies is somewhat cumbersome, since we assume so little about the actor model. Yet we should not be restricted to think we only have two options. The work of Zhao et al. [74] serves as an example of a mixture of the two. This suggests that similar mathematical structures (likely with a different blend of the two flavors) can be constructed to solve many similar problems.

Chapter 6

Conclusion

6.1 Summary of Results

This dissertation studies formal causality analysis of actor networks. I chose the tagged signal model as the mathematical foundation to analyze causality properties of actors. The tagged signal model is based on theory of posets. I defined the order union of posets, and connected it with graphical composition.

In Chapter 3, I presented a causality interface framework, which captures causality properties between an input and an output port. A causality interface is represented by an element in a dependency algebra, which is required to satisfy certain properties described in Section 3.1. These properties guarantee that the causality interfaces are composable. The general framework of causality interfaces does not define what the dependency algebra looks like. An appropriate choice of dependency algebra depends

on the model of computation of the actor network.

I next presented a dependency algebra for functional actors. Compositions of such causality interfaces were discussed. I presented an algorithm to verify liveness. In the case the tag set is totally ordered, I also presented an alternative theorem that states a necessary and sufficient condition for an actor network to be live. This allows verification to be performed independently and in parallel for each communication cycle. I also showed that causality analysis only needs to be performed for one port for each simple directed communication cycle.

The causality interface theory is applicable to a wide range of actor-oriented models. I gave examples of its application to timed models, which include synchronous languages, discrete-event and continuous-time models. Based on the framework of causality interfaces, I further defined causal and strict causal dependency of an output on an input port in timed systems. The theory is extended to verify whether a timed actor network is causal and live. I also discussed the application of the theory to dataflow models. Due to the Turing completeness of dataflow models, it is impossible to decide liveness in general. However, I showed that in many cases, liveness is decidable for dataflow models. Matthews [55] used a metric approach to give a sufficient condition for an actor network to be live. Using the causality interface formalism, I showed that Matthews' condition is sufficient but not necessary. Moreover, the condition given in the causality interface theory is much easier to verify.

In Chapter 5, I presented another framework called ordering dependency. The

ordering dependency model is more general in the sense that it captures causality properties that are missing in the functional abstraction of the actors. I showed how to use ordering dependencies to analyze rendezvous of sequential programs. Whether a rendezvous can be formed is equivalent to whether two posets are orderly unionable, or whether the composition of two graphs remains acyclic. I also showed that the PTIDES [74] programming model is an enrichment of the causality interface model, but a simplification of the ordering dependency model.

6.2 Future Work

6.2.1 Dynamic Causality Interfaces

Throughout the discussion in this dissertation, 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* [32]. In a modal model, an actor is associated with a state machine, and its interface can depend on the state of the state machine. 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 interface 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 [32]. Hence, a more precise analysis may not always be possible.

6.2.2 Determining Causality Interfaces for Atomic Actors

The causality analysis technique given in this dissertation 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 [30, 38, 58]) might be usable. However, given the Turing completeness of such lan-

guages, such analysis is likely to have to be conservative. A better alternative is probably to use an actor definition language such as Cal [28] or StreamIT [66] that is more amenable to such analysis.

Bibliography

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

[3] A. Aho, M. Garey, and J. Ullman. The transitive reduction of a directed graph. *SIAM Journal on Computing*, 1(2):131–737, 1972.

[4] Rajeev Alur and Thomas A. Henzinger. Reactive modules. *Formal Methods in System Design*, 15:7–48, 1999.

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

[6] Farhad Arbab. Abstract behavior types : A foundation model for components and their composition. *Science of Computer Programming*, 55:3–52, 2005.

[7] James R. Armstrong and F. Gail Gray. *VHDL Design Representation and Synthesis*. Prentice-Hall, second edition, 2000.

[8] John R. Barry, Edward A. Lee, and David G. Messerschmitt. *Digital Communication*. Kluwer Academic Publishers, 3rd edition, 2004.

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

[10] Albert Benveniste and Paul Le Guernic. Hybrid dynamical systems theory and the signal language. *IEEE Tr. on Automatic Control*, 35(5):525–546, 1990.

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

[12] Gerard Berry. *The Constructive Semantics of Pure Esterel*. Book Draft, 1996.

[13] Gerard Berry. The effectiveness of synchronous languages for the development of safety-critical systems. White paper, Esterel Technologies, 2003.

[14] G. Bilsen, M. Engels, R. Lauwereins, and J. A. Peperstraete. Static scheduling of multi-rate and cyclo-static DSP applications. In *Workshop on VLSI Signal Processing*. IEEE Press, 1994.

[15] Manfred Broy. Advanced component interface specification. In *Proceedings of the International Workshop on Theory and Practice of Parallel Programming (TPPP)*, pages 369–392, London, UK, 1994. Springer-Verlag.

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

[17] J. T. Buck. 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, 1993.

[18] Joseph T. Buck, Soonhoi Ha, Edward A. Lee, and David 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.

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

[20] James Adam Cataldo. Control algorithms for soft walls. Technical Report UCB/ERL M03/42, EECS Department, University of California, January 21 2004.

[21] Arindam Chakrabarti, Luca de Alfaro, and Thomas A. Henzinger. Resource

interfaces. In Rajeev Alur and Insup Lee, editors, *EMSOFT*, volume LNCS 2855, pages 117–133, Philadelphia, PA, 2003. Springer.

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

[23] B. A. Davey and H. A. Priestly. *Introduction to Lattices and Order*. Cambridge University Press, second edition, 2002.

[24] Luca de Alfaro and Thomas 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.

[25] E. A. de Kock, G. Essink, W. J. M. Smits, P. van der Wolf, J.-Y. Brunel, W.M. Kruijtzer, P. Lieverse, and K. A. Vissers. YAPI: Application modeling for signal processing systems. In *37th Design Automation Conference (DAC'00)*, pages 402–405, Los Angeles, CA, 2000.

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

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

[28] Johan Eker and Jrn 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.

[29] Johan Eker, Jrn W. Janneck, Edward A. Lee, Jie Liu, Xiaojun Liu, Jozsef Ludvig, Stephen Neuendorffer, Sonia Sachs, and Yuhong Xiong. Taming heterogeneity—the Ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, 2003.

[30] 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.

[31] Gene F. Franklin, J. David Powell, and Abbas Emami-Naeini. *Feedback Control of Dynamic Systems*. Addison-Wesley Publishing Company, 3rd edition, 1994.

[32] Alain Girault, Bilung 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):742–760, 1999.

[33] Gregor Göessler and Alberto Sangiovanni-Vincentelli. Compositional modeling in Metropolis. In *Second International Workshop on Embedded Software (EMSOFT)*, Grenoble, France, 2002. Springer-Verlag.

[34] Gregor Göessler and Joseph Sifakis. Composition for component-based modeling. *Science of Computer Programming*, 55, 2005.

[35] 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.

[36] Carl Hewitt. Viewing control structures as patterns of passing messages. *Journal of Artifical Intelligence*, 8(3):323–363, 1977.

[37] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8), 1978.

[38] Susan Horwitz, Thomas Reps, and David 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.

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

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

[41] Kurt Keutzer, Sharad Malik, A. Richard Newton, Jan Rabaey, and Alberto 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.

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

[43] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, September 1987.

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

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

[46] Edward 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.

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

[48] Edward A. Lee, Stephen Neuendorffer, and Michael J. Wirthlin. Actor-oriented design of embedded hardware and software systems. *Journal of Circuits, Systems, and Computers*, 12(3):231–260, 2003.

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

[50] Edward A. Lee and Yuhong Xiong. A behavioral type system and its application in Ptolemy II. *Formal Aspects of Computing Journal*, 16(3):210–237, 2004.

[51] Edward A. Lee and Haiyang Zheng. Operational semantics of hybrid systems. In Manfred Morari and Lothar Thiele, editors, *Hybrid Systems: Computation and Control (HSCC)*, volume LNCS 3414, pages 25–53, Zurich, Switzerland, 2005. Springer-Verlag.

[52] Jian Li and Petre Stoica. *Robust Adaptive Beamforming*. Wiley-Interscience, 2005.

[53] Xiaojun Liu. Semantic foundation of the tagged signal model. Ph.D. Thesis Technical Memorandum UCB/EECS-2005-31, EECS Department, University of California, Berkeley, December 20 2005.

[54] Xiaojun Liu and Edward A. Lee. CPO semantics of timed interactive actor networks. Technical Report UCB/EECS-2006-67, EECS Department, University of California, Berkeley, May 18 2006.

[55] S. G. Matthews. An extensional treatment of lazy data flow deadlock. *Theoretical Computer Science*, 151(1):195–205, 1995.

[56] Robin Milner. *A Calculus of Communicating Systems*. Springer Verlag, 1989.

[57] R. Neff and A. Zakhor. Dictionary approximation for matching pursuit video coding. In *International Conference on Image Processing*, Vancouver, Canada, September 2000.

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

[59] George A. Papadopoulos, Aristos Stavrou, and Odysseas Papapetrou. An implementation framework for software architectures based on the coordination paradigm. *Science of Computer Programming*, 60(1):27–67, 2006.

[60] Greg Papadopoulos and Farhad 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.

[61] Jonh G. Proakis. *Digital Communications*. McGraw-Hill, 4th edition, 2000.

[62] Jonh G. Proakis. *Digital Signal Processing*. Pearson Prentice Hall, 4th edition, 2007.

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

[64] 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.

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

[66] William Thies, Michal Karczmarek, and Saman Amarasinghe. StreamIt: A language for streaming applications. In *11th International Conference on Compiler Construction*, volume LNCS 2304, Grenoble, France, 2002. Springer-Verlag.

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

[68] W.W. Wadge. An extensional treatment of dataflow deadlock. *Theoretical Computer Science*, 13(1):3–15, 1981.

[69] Peter Wegner, Farhad Arbab, Dina Goldin, Peter McBurney, Michael Luck, and Dave Roberson. The role of agent interaction in models of computation (panel summary). In *Workshop on Foundations of Interactive Computation*, Edinburgh, 2005.

[70] G. Winskel. *The Formal Semantics of Programming Languages*. MIT Press, Cambridge, MA, USA, 1993.

[71] Michael Winter, Thomas Genssler, Alexander Christoph, Oscar Nierstrasz, Stéphane Ducasse, Roel Wuyts, Gabriela Arévalo, Peter Müller, Christian Stich, and Bastiaan Schönhage. 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, 2002.

[72] Yuhong 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.

[73] 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.

[74] Yang Zhao, Jie Liu, and Edward A. Lee. A programming model for time-synchronized distributed real-time systems. In *The 13th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 07)*, Bellevue, WA, USA, 2007.