4

The Semantics of Dataflow with Firing

Edward A. Lee

Eleftherios Matsikoudis
University of California, Berkeley

Abstract

Dataflow models of computation have intrigued computer scientists since the 1970s. They were first introduced by Jack Dennis as a basis for parallel programming languages and architectures, and by Gilles Kahn as a model of concurrency. Interest in these models of computation has been recently rekindled by the resurrection of parallel computing, due to the emergence of multicore architectures. However, Dennis and Kahn approached dataflow very differently. Dennis' approach was based on an operational notion of atomic firings driven by certain firing rules. Kahn's approach was based on a denotational notion of processes as continuous functions on infinite streams. This paper bridges the gap between these two points of view, showing that sequences of firings define a continuous Kahn process as the least fixed point of an appropriately constructed functional. The Dennis firing rules are sets of finite prefixes satisfying certain conditions that ensure determinacy. These conditions result in firing rules that are strictly more general than the blocking reads of the Kahn-MacQueen implementation of Kahn process networks, and solve some compositionality problems in the dataflow model[1].

4.1 Introduction

Three major variants of the dataflow model of computation have emerged in the literature: Kahn process networks [15], Dennis dataflow [10], and dataflow synchronous languages [2]. The first two are closely related, while the third is quite different. This paper deals only with the first two, which have a key difference. In Dennis dataflow, a process is implemented as an execution of atomic firings of actors. Although Dennis

dataflow can be viewed as a special case of Kahn process networks [18], the notion of firing has been absent from semantic models, which are most developed for Kahn process networks and dataflow synchronous languages.

Dennis and Kahn approach dataflow very differently. Dennis' approach is based on an operational notion of atomic firings driven by the satisfaction of firing rules. Kahn's approach is based on a denotational notion of processes as continuous functions on infinite streams. Dennis' approach influenced computer architecture [1, 26], compiler design, and concurrent programming languages [14]. Kahn's approach has influenced process algebras (see for example [6]) and semantics of concurrent systems (see for example [4, 20]). It has had practical realizations in stream languages [28] and operating systems (such as Unix pipes). Recently, interest in these models of computation has been rekindled by the resurrection of parallel computing, motivated by multicore architectures [8]. Dataflow models of computation are being explored for programming parallel machines [30], distributed systems [17, 21, 24], and embedded systems [19, 13]. Considerable effort is going into improved execution policies [31, 11, 32, 18] and standardization [22, 12].

This paper bridges the gap between Dennis and Kahn, showing that the methods pioneered by Kahn extend naturally to Dennis dataflow, embracing the notion of firing. This is done by establishing the relationship between a firing function and the Kahn process implemented as a sequence of firings of that function. A consequence of this analysis is a formal characterization of firing rules and firing functions that preserve determinacy.

4.2 Review of Kahn Process Networks

4.2.1 Ordered Sets

We begin with a brief review of ordered sets [9].

An *order relation* $\leqslant$ on a set A is a binary relation on A that is *reflexive* ($a \leqslant a'$), *transitive* (if $a \leqslant a'$ and $a' \leqslant a''$, then $a \leqslant a''$), and *antisymmetric* (if $a \leqslant a'$ and $a' \leqslant a$, then $a = a'$). Of course, we can define a corresponding *irreflexive* relation, denoted by $<$, with $a < a'$ if and only if $a \leqslant a'$ and $a \neq a'$. The structure $\langle A, \leqslant \rangle$ is an *ordered set*. If the order relation is *partial*, in the sense that there exist $a, a' \in A$ such that $a \not\leqslant a'$ and $a' \not\leqslant a$, then we will often refer to $\langle A, \leqslant \rangle$ as a *partially ordered set*, or simply a *poset*. If, on the other hand, the order relation

is *total*, in the sense that for all $a, a' \in A$, $a \leqslant a'$ or $a' \leqslant a$, then we will refer to $\langle A, \leqslant \rangle$ as a *totally ordered set*, or a *chain*.

For any ordered set $\langle A, \leqslant \rangle$ and any $B \subseteq A$, an element a is an *upper bound* of B in $\langle A, \leqslant \rangle$, iff for any $b \in B$, $b \leqslant a$. a is the *least upper bound* of B in $\langle A, \leqslant \rangle$ iff it is an upper bound of B, and for any other upper bound a' of B, $a \leqslant a'$. We write $\bigvee B$ to denote the least upper bound of B. The notion of *lower bound* and that of *greatest lower bound* are defined dually. In the case of two elements a_1 and a_2, we typically write $a_1 \vee a_2$ and $a_1 \wedge a_2$, instead of $\bigvee \{a_1, a_2\}$ and $\bigwedge \{a_1, a_2\}$. These are called the *join* and *meet* of a_1 and a_2.

A set $D \subseteq A$ is *directed* in $\langle A, \leqslant \rangle$ iff it is non-empty and every finite subset of D has an upper bound in $\langle A, \leqslant \rangle$. If every directed subset of A has a least upper bound in $\langle A, \leqslant \rangle$, then $\langle A, \leqslant \rangle$ is a *directed-complete* ordered set. If $\langle A, \leqslant \rangle$ is directed-complete and has a least element, then $\langle A, \leqslant \rangle$ is a *complete partial order*, or *cpo*. If $\langle A, \leqslant \rangle$ is directed-complete, and every non-empty subset of A has a greatest lower bound in $\langle A, \leqslant \rangle$, then $\langle A, \leqslant \rangle$ is a complete semilattice.

4.2.2 Sequences

We henceforth assume a non-empty set $\mathcal{V}$ of *values*. Each value represents a token, an atomic unit of data exchanged between the autonomous computing stations. We consider the set of all finite and infinite sequences over $\mathcal{V}$.

A *finite sequence of values*, or simply a *finite sequence*, is a function from the set $\{0, \ldots, n-1\}$ for some natural number n into the set $\mathcal{V}$. Notice that in the case of $n = 0$, $\{0, \ldots, n-1\} \to \mathcal{V} = \emptyset \to \mathcal{V} = \{\emptyset\}$. The empty set is therefore a finite sequence, which we call the *empty sequence* and denote by ε. We denote the set of all finite sequences of values by $\mathcal{V}^*$. This is of course the well known Kleene closure of the set $\mathcal{V}$.

An *infinite sequence of values*, or simply an *infinite sequence*, is a function from the set of all natural numbers ω into the set $\mathcal{V}$. We denote the set of all infinite sequences of values by $\mathcal{V}^\omega$. This is just another notation for the set $\omega \to \mathcal{V}$. We denote the set of all such sequences of values, finite or infinite, by $\mathcal{S}$; that is, $\mathcal{S} = \mathcal{V}^* \cup \mathcal{V}^\omega$.

For any finite sequence s, the *length* of s is the cardinal number of $\operatorname{dom} s$, which we denote by $|s|$. This is just the number of elements in s.

Informally, a sequence is just an ordered list of values. For any particular sequence s, we often list its values explicitly, writing $\langle v_0, v_1, \ldots, v_{|s|-1} \rangle$

if s is finite, and $\langle v_0, v_1, \ldots \rangle$ if s is infinite. If s is the empty sequence ε, then we simply write $\langle\ \rangle$.

A sequence s_1 is a *prefix* of a sequence s_2, and we write $s_1 \sqsubseteq s_2$, if and only if $s_1 \subseteq s_2$. We make use of the set-theoretic definition of function here, according to which the graph of a function is the function itself. We caution the reader not to misread our statement: not every subset of a sequence is a prefix. If that subset is a sequence, however, then it must be a prefix of the original sequence.

Informally, s_1 is a prefix of s_2 if and only if the first $|s_1|$ values of s_2 are the values of s_1 in the same order as in s_1; that is, for any natural number $i \in \operatorname{dom} s_1$, $s_2(i) = s_1(i)$.

The prefix relation $\sqsubseteq \subset \mathcal{S} \times \mathcal{S}$ is of course an order relation, and for any sequence s, $\varepsilon \sqsubseteq s$. The ordered set $\langle \mathcal{S}, \sqsubseteq \rangle$ is actually a complete semilattice. For any subset X of $\mathcal{S}$, we write $\bigsqcap X$ to denote the greatest lower bound of X in $\langle \mathcal{S}, \sqsubseteq \rangle$, namely the greatest common prefix of the sequences in X, and $\bigsqcup X$ to denote the least upper bound of X in $\langle \mathcal{S}, \sqsubseteq \rangle$, provided of course that this exists. In the case of two sequences s_1 and s_2, we typically write $s_1 \sqcap s_2$ and $s_1 \sqcup s_2$ for the meet and join of s_1 and s_2.

If s_1 is a finite sequence and s_2 an arbitrary sequence, then we write $s_1.s_2$ to denote the *concatenation* of s_1 and s_2. It is the unique sequence s with $\operatorname{dom} s = \{0, \ldots, |s_1| + |s_2| - 1\}$ if s_2 is finite, and $\operatorname{dom} s = \omega$ otherwise, such that for any $i \in \operatorname{dom} s$, $s(i) = s_1(i)$ if $i < |s_1|$, and $s(i) = s_2(i - |s_1|)$ otherwise.

Informally, $s_1.s_2$ is the result of appending the ordered list of values of s_2 right after the end of s_1. Note that any finite s_1 is a prefix of a sequence s iff there is a sequence s_2 such that $s_1.s_2 = s$. It should be clear that s_2 is unique.

4.2.3 Tuples of Sequences

A sequence of values models the traffic of tokens over a single communication line. A typical process network will have several communication lines, and a typical process will communicate over several of those. Thus, it will be useful to group together several different sequences and manipulate them as a single object. We do this using the notion of tuple.

A tuple is just a finite enumeration of objects. Here we are interested in tuples of sequences. For any natural number n, an *n-tuple of sequences*, or simply a *tuple*, is a function from $\{0, \ldots, n-1\}$ into $\mathcal{S}$. We let $\mathcal{S}^n$ denote the set of all n-tuples of sequences. For convenience,

we identify $\mathcal{S}^1$ with $\mathcal{S}$. Note that when $n = 0$, $\mathcal{S}^n = \emptyset \to \mathcal{S} = \{\emptyset\}$. The empty set is thus a tuple, which we call the *empty tuple*.

We use boldface letters to denote tuples. If $\boldsymbol{s}$ is an n-tuple, then for any $i \in \{0, \ldots, n-1\}$, we often write $\boldsymbol{s}_i$ instead of $\boldsymbol{s}(i)$. Also, we often list the sequences within a tuple explicitly, writing $\langle s_0, \ldots, s_{n-1} \rangle$.

We say that an n-tuple is *finite* if and only if for any $i \in \{0, \ldots, n-1\}$, s_i is a finite sequence. This is of course vacuously true for the empty tuple.

The prefix order on sequences induces an order on n-tuples for any fixed n. The order we have in mind is the pointwise order. We say that an n-tuple $\boldsymbol{s}_1$ is a *prefix* of an n-tuple $\boldsymbol{s}_2$, and we write $\boldsymbol{s}_1 \sqsubseteq \boldsymbol{s}_2$, if and only if for any $i \in \{0, \ldots, n-1\}$, $\boldsymbol{s}_1(i) \sqsubseteq \boldsymbol{s}_2(i)$. Notice that the n-tuple of empty sequences, denoted by $\boldsymbol{\varepsilon}_n$, is a prefix of every other n-tuple. The ordered set $\langle \mathcal{S}^n, \sqsubseteq \rangle$ is a complete semilattice, where infima and suprema are calculated pointwise, simply because $\langle \mathcal{S}, \sqsubseteq \rangle$ is a complete semilattice itself.

If $\boldsymbol{s}_1$ is a finite n-tuple and $\boldsymbol{s}_2$ an arbitrary n-tuple, then we write $\boldsymbol{s}_1.\boldsymbol{s}_2$ to denote the pointwise concatenation of $\boldsymbol{s}_1$ and $\boldsymbol{s}_2$. It is the unique n-tuple $\boldsymbol{s}$ such that for any $i \in \{0, \ldots, n-1\}$, $\boldsymbol{s}(i) = \boldsymbol{s}_1(i).\boldsymbol{s}_2(i)$.

When $n = 0$, $\mathcal{S}^n$ has only one element, the empty tuple $\emptyset$. Hence, it must be the case that $\emptyset.\emptyset = \emptyset$. This is precisely what the pointwise concatenation evaluates to. Note again that for any finite $\boldsymbol{s}_1$, $\boldsymbol{s}_1 \sqsubseteq \boldsymbol{s}$ if and only if there is some tuple $\boldsymbol{s}_2$ such that $\boldsymbol{s}_1.\boldsymbol{s}_2 = \boldsymbol{s}$, in which case, this tuple $\boldsymbol{s}_2$ is unique.

4.2.4 Kahn Processes

Before we can formalize the notion of a process, we must review a technical condition that we will need to impose.

A function $F : \mathcal{S}^m \to \mathcal{S}^n$ is *monotone* if and only if for all $\boldsymbol{s}_1, \boldsymbol{s}_2 \in \mathcal{S}^m$,

$$\boldsymbol{s}_1 \sqsubseteq \boldsymbol{s}_2 \implies F(\boldsymbol{s}_1) \sqsubseteq F(\boldsymbol{s}_2).$$

Informally, feeding a computing station that realizes a monotone function with additional input can only cause it to produce additional output. This is really a notion of causality, in that "future input concerns only future output" (see [15]).

A function $F : \mathcal{S}^m \to \mathcal{S}^n$ is *Scott-continuous*, or simply *continuous*, if and only if F is monotone, and for any subset D of $\mathcal{S}^m$ that is directed in $\langle \mathcal{S}^m, \sqsubseteq \rangle$,

$$F(\bigsqcup D) = \bigsqcup \{F(\boldsymbol{s}) \mid \boldsymbol{s} \in D\}.$$

Notice here that since F is monotone, the set $\{F(\boldsymbol{s}) \mid \boldsymbol{s} \in D\}$ is itself directed in $\langle \mathcal{S}^n, \sqsubseteq \rangle$, and hence has a least upper bound therein.

In order to better understand the importance of this notion, we must take notice of the additional structure that our ordered sets have. For any natural number m, the complete semilattice $\langle \mathcal{S}^m, \sqsubseteq \rangle$ is *algebraic*: for every $\boldsymbol{s} \in \mathcal{S}^m$,

$$\boldsymbol{s} = \bigsqcup \{\boldsymbol{s}' \sqsubseteq \boldsymbol{s} \mid \boldsymbol{s}' \text{ is finite}\}.$$

The set $\{\boldsymbol{s}' \sqsubseteq \boldsymbol{s} \mid \boldsymbol{s}' \text{ is finite}\}$ is of course directed in $\langle \mathcal{S}^m, \sqsubseteq \rangle$. Hence, we can obtain every m-tuple as the least upper bound of a set of finite tuples that is directed in $\langle \mathcal{S}^m, \sqsubseteq \rangle$. The response of a continuous function to an input tuple is therefore completely defined by its responses to the finite prefixes of that tuple. This is really a computability notion, in that a computing station cannot churn out some output only after it has received an infinite amount of input.

We remark here that continuity in this context is exactly the topological notion of continuity in a particular topology, which is called the *Scott topology*. In this topology, the set of all tuples with a particular finite prefix is an open set, and the collection of all these sets is a base for the topology.

A *Kahn process*, or just a *process*, is a continuous function $F : \mathcal{S}^m \to \mathcal{S}^n$ for some m and n. If $m = 0$, then we say that F is a *source*; $\mathcal{S}^m = \mathcal{S}^0 = \{\emptyset\}$ has a single member, the empty tuple, and hence F is trivially constant. If $n = 0$, then we say that F is a *sink*. In either case, F is trivially continuous.

Not every monotone function is continuous, and thus a Kahn process. For instance, consider a function $F : \mathcal{S} \to \mathcal{S}$ such that for any sequence s,

$$F(s) = \begin{cases} \langle\ \rangle & \text{if } \boldsymbol{s} \text{ is finite;} \\ \langle v \rangle & \text{otherwise.} \end{cases}$$

Here v is some arbitrary value. It is easy to verify that F is monotone but not continuous.

For an example of a continuous function, consider the *unit delay* process $D_v : \mathcal{S} \to \mathcal{S}$, such that for any sequence s,

$$D_v(s) = \langle v \rangle.s, \tag{4.1}$$

where v is an arbitrary but fixed value. The effect of this process is to output an initial token of value v before starting to churn out the tokens

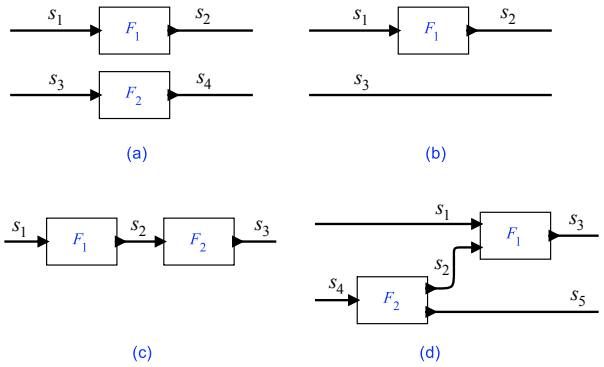

Fig. 4.1. Examples of compositions of processes.

arriving at its input, in the same order in which they arrive. We will have more to say about the unit delay below.

4.2.5 Compositions of Kahn Processes and Determinacy

A finite composition of Kahn processes is a collection $\{s_1, \ldots, s_p\}$ of sequences and a collection $\{F_1, \ldots, F_q\}$ of processes relating them, such that no sequence is the output of more than one process. Any sequence that is not the output of any of the functions is an input to the composition.

A composition is determinate if and only if given the input sequences, all other sequences are uniquely determined. Obviously, a Kahn process by itself is determinate, since it is a functional mapping from input sequences to output sequences.

Examples of finite compositions of Kahn processes are shown in Figure 4.1. In each of these examples, given the component processes, it is obvious how to construct a processes that maps the input sequences (those that are not outputs of any process) to the other sequences. Each of these compositions is thus determinate. Following Broy [5], we can iteratively compose processes using patterns like those in Figure 4.1 to argue that arbitrary compositions are determinate. The most challenging part of this strategy is to handle feedback. (An alternative approach to this composition problem is given by Stark [27]).

Feedback compositions of Kahn processes may or may not be determinate. Consider for example the identity function I, such that for any sequence s, $I(s) = s$. I is trivially continuous, and thus a Kahn process.

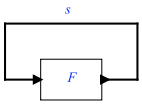

Fig. 4.2. Feedback (a directed self-loop).

Suppose that we form a very simple composition of the identity process by feeding back the output to the input, letting $F = I$ in Figure 4.2. There are no inputs to the composition, which is therefore determinate if and only if the sequence s is uniquely determined. However, any sequence s satisfies the constraint of the composition, so it is not uniquely determined.

4.2.6 Least-Fixed-Point Semantics

There is an alternative interpretation due to Kahn [15] that makes the example in Figure 4.2 determinate. Under this interpretation, any process composition is determinate. Moreover, this interpretation is consistent with the execution policies often used for such systems (their operational semantics), and hence it is an entirely reasonable denotational semantics for the composition. This interpretation is known as the least-fixed-point semantics, and in particular as the Kahn principle.

The Kahn principle is based on a well-known fixed-point theorem stating that a continuous function $F : X \to X$ on a cpo $\langle X, \leqslant \rangle$ has a least fixed point x in $\langle X, \leqslant \rangle$; that is, there is an $x \in X$ such that $F(x) = x$, and for any other $y \in X$ for which $F(y) = y$, $x \leqslant y$. Furthermore, the theorem is constructive, providing an algorithmic procedure for finding the least fixed point: the least fixed point of F is the least upper bound of all finite iterations of F starting from the least element in $\langle X, \leqslant \rangle$.

To put it into our context, suppose that $F : \mathcal{S}^n \to \mathcal{S}^n$ is a process, and consider the following sequence of n-tuples:

$$\boldsymbol{s}_0 = \boldsymbol{\varepsilon}_n, \ \boldsymbol{s}_1 = F(\boldsymbol{s}_0), \ \boldsymbol{s}_2 = F(\boldsymbol{s}_1), \ \ldots . \tag{4.2}$$

Since F is monotone, and the tuple of empty sequences $\boldsymbol{\varepsilon}_n$ is a prefix of any other n-tuple, $\boldsymbol{s}_i \sqsubseteq \boldsymbol{s}_j$ if and only if $i \leq j$. Hence, $\{\boldsymbol{s}_0, \boldsymbol{s}_1, \ldots\}$ is a chain, and thus directed in $\langle \mathcal{S}^n, \sqsubseteq \rangle$, and since the latter is directed-complete, $\{\boldsymbol{s}_0, \boldsymbol{s}_1, \ldots\}$ has a least upper bound in $\langle \mathcal{S}^n, \sqsubseteq \rangle$. The fixed-point theorem states that this least upper bound is the least fixed point of F.

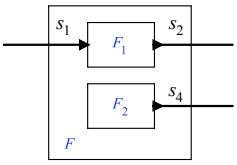

Fig. 4.3. Composition with a source of an infinite sequence.

This theorem is quite similar to the well-known Knaster-Tarski fixed-point theorem, which applies to complete lattices rather than complete partial orders. For this reason, this approach to semantics is sometimes called Tarskian. The application of the theorem to programming language semantics was pioneered by Scott [25]. However, Kahn [15] was the first to recognize its potential in modeling and design of complex distributed systems.

Under this least-fixed-point principle, the value of s in Figure 4.2 is uniquely determined as the empty sequence ε when F is the identity process I. This is consistent with our intuition; the identity process will not produce an output token, unless there is some input token to cause it to.

Notice that (4.2) might suggest a reasonable execution policy for a network: start with every sequence empty, and begin iterating the evaluation of every process. In the limit, every sequence will converge to the least fixed point of the composite process, in accordance with the interpretation suggested by the Kahn principle.

4.2.7 Practical Issues

There are serious practical problems with using (4.2) as an execution policy. If any process in the composition evaluates to an infinite tuple at some stage of the iteration, then the execution of that process will never terminate, and thus preclude the progress of the iteration. This will happen immediately in a composition like the one in Figure 4.3, where the process F_2 is a source of an infinite sequence.

In practice, we need to partially evaluate processes, carefully controlling the length of each sequence. The problem is addressed by Parks [23], who devises a general strategy to avoid accumulating unbounded numbers of unconsumed tokens, whenever it is possible to do so. All partially evaluated sequences are guaranteed to be prefixes of the sequences corresponding to the denotational semantics of the process composition

(although, as pointed out in [11], there is no assurance of convergence to those sequences, which may not be desirable anyway).

4.3 Dataflow with Firing

4.3.1 Dataflow Actors

We begin with a simple definition and generalize later. Our first attempt will serve as a gentle introduction, and help motivate the need for the more general case.

A *dataflow actor*, or dimply an *actor*, with m *inputs* and n *outputs* is a pair $\langle R, f \rangle$, where

(i) R is a set of finite m-tuples;
(ii) $f : \mathcal{S}^m \to \mathcal{S}^n$ is a (possibly partial) function defined at least on R;
(iii) $f(\boldsymbol{r})$ is finite for every $\boldsymbol{r} \in R$;
(iv) for all $\boldsymbol{r}, \boldsymbol{r}' \in R$, if $\boldsymbol{r} \neq \boldsymbol{r}'$, then $\{\boldsymbol{r}, \boldsymbol{r}'\}$ does not have an upper bound in $\langle \mathcal{S}^m, \sqsubseteq \rangle$.

We call each $\boldsymbol{r} \in R$ a *firing rule*, and f the *firing function* of the actor.

The last condition is equivalent to the following statement: for any given m-tuple $\boldsymbol{s}$, there is at most one firing rule $\boldsymbol{r}$ in R such that $\boldsymbol{r} \sqsubseteq \boldsymbol{s}$. We remark here that because $\langle \mathcal{S}^m, \sqsubseteq \rangle$ is a complete semilattice, $\boldsymbol{r}$ and $\boldsymbol{r}'$ have an upper bound in $\langle \mathcal{S}^m, \sqsubseteq \rangle$ if and only they have a least upper bound in $\langle \mathcal{S}^m, \sqsubseteq \rangle$, or alternatively, if their join $\boldsymbol{r} \sqcap \boldsymbol{r}'$ is defined.

If $m = 0$, then R is a subset of the singleton set $\{\emptyset\}$, and condition (iv) is trivially satisfied. If $n = 0$, then condition (iii) is trivially satisfied.

4.3.2 Dataflow Processes

Let $\langle R, f \rangle$ be a dataflow actor with m inputs and n outputs. We want to define a Kahn process $F : \mathcal{S}^m \to \mathcal{S}^n$ based on this actor, and a reasonable condition to impose is that for any m-tuple $\boldsymbol{s}$,

$$F(\boldsymbol{s}) = \begin{cases} f(\boldsymbol{r}).F(\boldsymbol{s}') & \text{if there exists } \boldsymbol{r} \in R \text{ such that } \boldsymbol{s} = \boldsymbol{r}.\boldsymbol{s}'; \\ \boldsymbol{\varepsilon}_n & \text{otherwise.} \end{cases} \tag{4.3}$$

Of course, this is not a definition. It is by no means obvious that such an F exists, nor that this F is unique, or even a process. Nonetheless, it is possible to use the least-fixed-point principle to resolve these issues, and turn (4.3) into a proper definition. But before we can do this, we will

need to review some order-theoretic facts about functions over tuples of sequences.

For fixed m and n, we write $\mathcal{S}^m \to \mathcal{S}^n$ to denote the set of all functions from $\mathcal{S}^m$ into $\mathcal{S}^n$. The prefix order on n-tuples induces a pointwise order on this set. We shall say that the function $F : \mathcal{S}^m \to \mathcal{S}^n$ is a *prefix* of the function $G : \mathcal{S}^m \to \mathcal{S}^n$, and write $F \sqsubseteq G$, if and only if $F(\boldsymbol{s}) \sqsubseteq G(\boldsymbol{s})$ for any m-tuple $\boldsymbol{s}$. Notice that the function $\boldsymbol{s} \mapsto \boldsymbol{\varepsilon}_n$ mapping every m-tuple $\boldsymbol{s}$ to the n-tuple of empty sequences is a prefix of any other function in the set. The ordered set $\langle \mathcal{S}^m \to \mathcal{S}^n, \sqsubseteq \rangle$ is a complete semilattice, simply because $\langle \mathcal{S}^n, \sqsubseteq \rangle$ is a complete semilattice. But for our purposes here, is suffices to know that $\langle \mathcal{S}^m \to \mathcal{S}^n, \sqsubseteq \rangle$ is a cpo.

Now consider the functional $\phi : (\mathcal{S}^m \to \mathcal{S}^n) \to (\mathcal{S}^m \to \mathcal{S}^n)$ associated with the actor $\langle R, f \rangle$, defined such that for any $F \in \mathcal{S}^m \to \mathcal{S}^n$ and any m-tuple $\boldsymbol{s}$,

$$\phi(F)(\boldsymbol{s}) = \begin{cases} f(\boldsymbol{r}).F(\boldsymbol{s}') & \text{if there exists } \boldsymbol{r} \in R \text{ such that } \boldsymbol{s} = \boldsymbol{r}.\boldsymbol{s}'; \\ \boldsymbol{\varepsilon}_n & \text{otherwise.} \end{cases} \tag{4.4}$$

Theorem 4.3.1 *ϕ is monotone.*

Proof Let F_1 and F_2 be arbitrary functions of type $\mathcal{S}^m \to \mathcal{S}^n$, and suppose that $F_1 \sqsubseteq F_2$.

If there is a firing rule $\boldsymbol{r} \in R$ such that $\boldsymbol{r} \sqsubseteq \boldsymbol{s}$, then by condition (iv), $\boldsymbol{r}$ is unique, and hence $\phi(F_1)(\boldsymbol{s}) = f(\boldsymbol{r}).F_1(\boldsymbol{s}')$ and $\phi(F_2)(\boldsymbol{s}) = f(\boldsymbol{r}).F_2(\boldsymbol{s}')$, where $\boldsymbol{s} = \boldsymbol{r}.\boldsymbol{s}'$. However, by assumption, $F_1(\boldsymbol{s}') \sqsubseteq F_2(\boldsymbol{s}')$ for any m-tuple $\boldsymbol{s}'$, and hence $\phi(F_1)(\boldsymbol{s}) \sqsubseteq \phi(F_2)(\boldsymbol{s})$.

Otherwise, $\phi(F_1)(\boldsymbol{s}) = \boldsymbol{\varepsilon}_n = \phi(F_2)(\boldsymbol{s})$.

In either case, $\phi(F_1)(\boldsymbol{s}) \sqsubseteq \phi(F_2)(\boldsymbol{s})$, and hence ϕ is monotone. □

Since ϕ is a monotone function over the cpo $\langle \mathcal{S}^m \to \mathcal{S}^n, \sqsubseteq \rangle$, it has a least fixed point F in $\langle \mathcal{S}^m \to \mathcal{S}^n, \sqsubseteq \rangle$ [9], which must satisfy (4.3). This is reassuring, but we can actually go a step further, and give a constructive procedure for finding that least fixed point.

Theorem 4.3.2 *ϕ is continuous.*

Proof Let $D \subseteq \mathcal{S}^m \to \mathcal{S}^n$ be directed in $\langle \mathcal{S}^m \to \mathcal{S}^n, \sqsubseteq \rangle$, and $\boldsymbol{s}$ an arbitrary m-tuple.

If there is a firing rule $\boldsymbol{r} \in R$ such that $\boldsymbol{r} \sqsubseteq \boldsymbol{s}$, then by condition (iv),

$\boldsymbol{r}$ is unique, and hence for every $F \in \mathcal{S}^m \to \mathcal{S}^n$, $\phi(F)(\boldsymbol{s}) = f(\boldsymbol{r}).F(\boldsymbol{s}')$, where $\boldsymbol{s} = \boldsymbol{r}.\boldsymbol{s}'$. Thus,

$$\begin{aligned}\bigsqcup\{\phi(F)(\boldsymbol{s}) \mid F \in D\} &= \bigsqcup\{f(\boldsymbol{r}).F(\boldsymbol{s}') \mid F \in D\} \\ &= f(\boldsymbol{r}).\bigsqcup\{F(\boldsymbol{s}') \mid F \in D\} \\ &= f(\boldsymbol{r}).(\bigsqcup D)(\boldsymbol{s}') \\ &= \phi(\bigsqcup D)(\boldsymbol{s}).\end{aligned}$$

Notice that since D is directed in $\langle \mathcal{S}^m \to \mathcal{S}^n, \sqsubseteq\rangle$, it has a least upper bound therein, $\langle \mathcal{S}^m \to \mathcal{S}^n, \sqsubseteq\rangle$ being a cpo.

Otherwise, for every $F \in \mathcal{S}^m \to \mathcal{S}^n$, $\phi(F)(\boldsymbol{s}) = \boldsymbol{\varepsilon}_n$, and hence

$$\bigsqcup\{\phi(F)(\boldsymbol{s}) \mid F \in D\} = \boldsymbol{\varepsilon}_n = \phi(\bigsqcup D)(\boldsymbol{s}).$$

In either case,

$$\bigsqcup\{\phi(F)(\boldsymbol{s}) \mid F \in D\} = \phi(\bigsqcup D)(\boldsymbol{s}),$$

and hence ϕ is continuous. □

Since ϕ is continuous, not only does it have a least fixed point, but there is a constructive procedure for finding that least fixed point [9]. We can start with the least element in $\langle \mathcal{S}^m \to \mathcal{S}^n, \sqsubseteq\rangle$, the function $\boldsymbol{s} \mapsto \boldsymbol{\varepsilon}_n$ mapping every m-tuple $\boldsymbol{s}$ to the empty sequence, and iterate ϕ to obtain the following sequence of functions:

$$F_0 = \boldsymbol{s} \mapsto \boldsymbol{\varepsilon}_n,\ F_1 = \phi(F_1),\ F_2 = \phi(F_1),\ \ldots. \tag{4.5}$$

Since ϕ is monotone, and $\boldsymbol{s} \mapsto \boldsymbol{\varepsilon}_n$ is a sequence of every other function, the set $\{F_0, F_1, \ldots\}$ is a chain, and hence directed in $\langle \mathcal{S}^m \to \mathcal{S}^n, \sqsubseteq\rangle$. Thus, it has a least upper bound therein, which is the least fixed point of ϕ.

Let us examine this chain more closely for some fixed m-tuple $\boldsymbol{s}$. Suppose that there is some sequence of firing rules $\langle \boldsymbol{r}_1, \boldsymbol{r}_2, \ldots\rangle$ such that $\boldsymbol{s} = \boldsymbol{r}_1.\boldsymbol{r}_2.\ldots$. Then, for this particular m-tuple, we can rewrite (4.5) in the following form:

$$\begin{aligned}F_0(\boldsymbol{s}) &= \boldsymbol{\varepsilon}_n \\ F_1(\boldsymbol{s}) &= f(\boldsymbol{r}_1) \\ F_2(\boldsymbol{s}) &= f(\boldsymbol{r}_1).f(\boldsymbol{r}_2) \\ &\ldots\end{aligned} \tag{4.6}$$

This is an exact description of the operational semantics in Dennis dataflow, with respect to a single actor. Start with the actor producing

only the empty sequence. Then find the prefix of the input that matches a firing rule, and invoke the firing function on that prefix, producing a partial output. Notice here that because of condition (iv), no more than one firing rule can match a prefix of the input at any time. Then find the prefix of the remaining input that matches another firing rule, invoke the firing function on that prefix, and concatenate the result with the output.

In general, even when $\boldsymbol{s}$ is infinite, it is possible that there is only a finite sequence of firing rules $\langle \boldsymbol{r}_0, \ldots, \boldsymbol{r}_p \rangle$ such that $\boldsymbol{s} = \boldsymbol{r}_0 . \ldots . \boldsymbol{r}_p . \boldsymbol{s}'$, with $\boldsymbol{s}'$ having no prefix in R. In both the operational semantics of Dennis dataflow and the denotational interpretation of (4.6), the firings simply stop, and the output is finite.

When $m = 0$, the least fixed point of ϕ is a source process, and if $\emptyset \in R$, then it produces the sequence $f(\emptyset).f(\emptyset).\cdots$. If $f(\emptyset)$ is non-empty, then this is infinite and periodic. This might seem limiting for dataflow processes that act as sources, but in fact it is not; a source with a more complicated output sequence can be constructed using a feedback composition, as in Figure 4.2.

When $n = 0$, the least fixed point of ϕ is a sink process, producing the sequence $\emptyset.\emptyset.\cdots = \emptyset$.

In view of this perfect coincidence with the operational semantics, we are tempted to define a Kahn process based on the actor $\langle R, f \rangle$ as this least fixed point of ϕ. But in order to do this, we still need to prove that in the general case, this least fixed point of ϕ is actually a continuous function, and thus a Kahn process. It suffices to prove the following theorem:

Theorem 4.3.3 *For any $F : \mathcal{S}^m \to \mathcal{S}^n$, if F is continuous, then $\phi(F)$ is also continuous.*

Proof Let $F : \mathcal{S}^m \to \mathcal{S}^n$ be a continuous function, and $D \subseteq \mathcal{S}^m$ directed in $\langle \mathcal{S}^m, \sqsubseteq \rangle$.

Suppose, toward contradiction, that there are $\boldsymbol{r}_1, \boldsymbol{r}_2 \in R$ and $\boldsymbol{s}_1, \boldsymbol{s}_2 \in D$ such that $\boldsymbol{r}_1 \neq \boldsymbol{r}_2$, but $\boldsymbol{r}_1 \sqsubseteq \boldsymbol{s}_1$ and $\boldsymbol{r}_2 \sqsubseteq \boldsymbol{s}_2$. Then since D is directed in $\langle \mathcal{S}^m, \sqsubseteq \rangle$, $\{\boldsymbol{s}_1, \boldsymbol{s}_2\}$ has an upper bound in D, which is also an upper bound of $\{\boldsymbol{r}_1, \boldsymbol{r}_2\}$, in contradiction to (iv).

Therefore, there is at most one $\boldsymbol{r} \in R$ that is a prefix of some tuple in D.

If there is such an $\boldsymbol{r} \in R$, then

$$\begin{aligned}\bigsqcup\{\phi(F)(\boldsymbol{s}) \mid \boldsymbol{s} \in D\} &= \bigsqcup\{f(\boldsymbol{r}).F(\boldsymbol{s}') \mid \boldsymbol{r}.\boldsymbol{s}' \in D\} \\ &= f(\boldsymbol{r}).\bigsqcup\{F(\boldsymbol{s}') \mid \boldsymbol{r}.\boldsymbol{s}' \in D\} \\ &= f(\boldsymbol{r}).F(\bigsqcup\{\boldsymbol{s}' \mid \boldsymbol{r}.\boldsymbol{s}' \in D\}) \\ &= \phi(F)(\bigsqcup D).\end{aligned}$$

Notice that since D is directed in $\langle \mathcal{S}^m, \sqsubseteq \rangle$, $\{\boldsymbol{s}' \mid \boldsymbol{r}.\boldsymbol{s}' \in D\}$ is also directed in $\langle \mathcal{S}^m, \sqsubseteq \rangle$, and in particular, $\boldsymbol{r}.\bigsqcup\{\boldsymbol{s}' \mid \boldsymbol{r}.\boldsymbol{s}' \in D\} = \bigsqcup D$.

Otherwise, there is no firing rule in R that is a prefix of some tuple in D, and hence

$$\bigsqcup\{\phi(F)(\boldsymbol{s}) \mid \boldsymbol{s} \in D\} = \boldsymbol{\varepsilon}_n = \phi(F)(\bigsqcup D).$$

In either case,

$$\bigsqcup\{\phi(F)(\boldsymbol{s}) \mid \boldsymbol{s} \in D\} = \phi(F)(\bigsqcup D),$$

and hence $\phi(F)$ is continuous. □

Since $\boldsymbol{s} \mapsto \boldsymbol{\varepsilon}_n$ is trivially continuous, and continuous functions are closed under pointwise suprema [9], an easy induction suffices to see that the least fixed point of ϕ is a continuous function. Note here that the firing function f need not be continuous. In fact, it does not even need to be monotone. The continuity of the least fixed point of ϕ is guaranteed if $\langle R, f \rangle$ is a valid actor description according to conditions (i) through (iv).

4.3.3 Examples of Firing Rules

Consider a system where the set of token values is $\mathcal{V} = \{0, 1\}$. Let us examine some possible sets $R \subset \mathcal{S}$ of firing rules for unary firing functions $f : \mathcal{S} \to \mathcal{S}$.

The following sets of firing rules all satisfy condition (iv) above:

$$\begin{gathered}\{\langle\ \rangle\}; \\ \{\langle 0 \rangle\}; \\ \{\langle 0 \rangle, \langle 1 \rangle\}; \\ \{\langle 0, 0 \rangle, \langle 0, 1 \rangle, \langle 1, 0 \rangle, \langle 1, 1 \rangle\}.\end{gathered} \tag{4.7}$$

The first of these corresponds to a function that consumes no tokens from its input sequence, and can fire infinitely regardless of the length of the input sequence. The second consumes only the leading zeros

from the input sequence, and then stops firing. The third consumes one token from the input on every firing, regardless of its value. The fourth consumes two tokens on the input on every firing, again regardless of the values.

An example of a set of firing rules that does not satisfy condition (iv) is:

$$\{\langle\ \rangle, \langle 0\rangle, \langle 1\rangle\}. \tag{4.8}$$

Such firing rules would correspond to an actor that could nondeterministically consume or not consume an input token upon firing.

The firing rules in (4.8) would also correspond to the firing rules of the unit delay defined in (4.1), so such a process cannot be a dataflow actor under this definition. In fact, delays in dataflow actor networks are usually implemented directly as initial tokens on an arc. Thus, if we admit such an implementation, then there is no loss of generality here. The implementation cost is lower, and this strategy avoids having to have special firing rules for delays that, if allowed in general, could introduce non-determinism. Furthermore, once we admit this sort of implementation for the unit delay, it is easy to model arbitrary actors with state using a single self-loop initialized to their initial state.

Let us examine now some possible sets $R \subset \mathcal{S}^2$ of firing rules for binary firing functions $f : \mathcal{S}^2 \to \mathcal{S}$.

The following sets of firing rules all satisfy condition (iv):

$$\begin{gathered}\{\langle\langle 0\rangle, \langle 0\rangle\rangle, \langle\langle 0\rangle, \langle 1\rangle\rangle, \langle\langle 1\rangle, \langle 0\rangle\rangle, \langle\langle 1\rangle, \langle 1\rangle\rangle\}; \\ \{\langle\langle 0\rangle, \langle\ \rangle\rangle, \langle\langle 1\rangle, \langle 0\rangle\rangle, \langle\langle 1\rangle, \langle 1\rangle\rangle\}; \\ \{\langle\langle 0\rangle, \langle\ \rangle\rangle, \langle\langle 1\rangle, \langle\ \rangle\rangle\}.\end{gathered} \tag{4.9}$$

The first of these corresponds to an actor that consumes one input token from each of its inputs. For example, this could implement a logic function such as AND or OR. The second corresponds to a conditional actor, where the first input provides a control token on every firing. If the control token has value '1', then a token is consumed from the second input. Otherwise, no token is consumed from the second input. The third corresponds to an actor that has effectively one input, never consuming a token from the second input.

The following set of firing rules does not satisfy condition (iv):

$$\{\langle\langle 0\rangle, \langle\ \rangle\rangle, \langle\langle 1\rangle, \langle\ \rangle\rangle, \langle\langle\ \rangle, \langle 0\rangle\rangle, \langle\langle\ \rangle, \langle 1\rangle\rangle\}. \tag{4.10}$$

These would be the firing rules of a non-determinate merge, a process that can consume a token on either input and copy it to its output. The

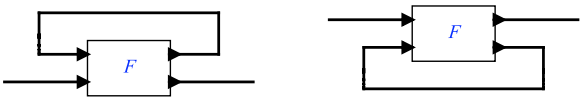

Fig. 4.4. If F is an identity process, the appropriate firing rules are (4.10).

non-determinate merge is not a monotone process, and so use of it in a Kahn process network could result in non-determinism.

It is interesting to notice that the sets of firing rules of (4.7) and (4.9) can all be implemented in a blocking-read fashion, according to the Kahn-MacQueen implementation of Kahn process networks [16]. An example of a process that cannot be implemented using blocking reads has the firing rules:

$$\{\langle\langle 1\rangle, \langle 0\rangle, \langle\ \rangle\rangle, \langle\langle 0\rangle, \langle\ \rangle, \langle 1\rangle\rangle, \langle\langle\ \rangle, \langle 0\rangle, \langle 1\rangle\rangle\}. \tag{4.11}$$

These firing rules satisfy (iv) and correspond to the Gustave function [3], a function defining a process which is stable, but not sequential as the other examples.

While actors that satisfy conditions (i) through (iv) above yield continuous Kahn processes, these conditions are somewhat more restrictive than what is really necessary. The firing rules in (4.10), for example, are not only the firing rules for the dangerous non-determinate merge, but also the firing rules for a perfectly harmless two-input two-output identity process. At first glance, it might seem that this sort of identity process could be implemented using the first set of firing rules of (4.9), though this will not work. The two examples in Figure 4.4 show why not. In the first example, the first (top) input and output should be the empty sequence under the least-fixed-point semantics, so there will never be a token to trigger any firing rule of (4.9). In the second of these examples, the second (bottom) input and output present the same problem. The firing rules of (4.10), however, have no difficulty with these cases. We next replace condition (iv) with a more general rule that solves such problems.

4.3.4 Commutative Firings

Many dataflow models having a notion of firing are not compositional. These compositionality issues are discussed in a very general framework by Talcott [29]. In our context, the problem is simply that an aggregation of actors that can be individually described using firing rules and firing

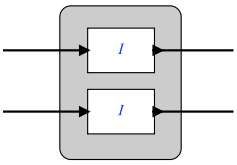

Fig. 4.5. A two-input two-output identity process described as an aggregation of two one-input one-output identity processes.

functions cannot be collectively described in this way. This problem was alluded to in the final example of the last subsection, which is the simplest example illustrating the problem. It is possible to think of a two-input two-output identity process as an aggregation of two one-input one-output identity processes, as in Figure 4.5. One-input one-output identity processes are trivially described as actors that satisfy conditions (i) through (iv), but a two-input two-output identity process cannot be so described.

In order to solve this problem, we replace condition (iv) with the following more elaborate condition:

(iv′) for all $\boldsymbol{r}, \boldsymbol{r}' \in R$, if $\boldsymbol{r} \neq \boldsymbol{r}'$ and $\{\boldsymbol{r}, \boldsymbol{r}'\}$ has an upper bound in $\langle \mathcal{S}^m, \sqsubseteq \rangle$, then $f(\boldsymbol{r}).f(\boldsymbol{r}') = f(\boldsymbol{r}').f(\boldsymbol{r})$ and $\boldsymbol{r} \sqcap \boldsymbol{r}' = \boldsymbol{\varepsilon}_m$.

This condition states that if any two firing rules are consistent, namely they have a common upper bound, and therefore can possibly be enabled at the same time, then it makes no difference in what order we use these firing rules; the values of the firing function at these consistent rules commute with respect to the concatenation operator. Furthermore, any two consistent firing rules have no common prefix other than the m-tuple of empty sequences.

It is easy to see that when condition (iv′) is satisfied,

$$\boldsymbol{r} \sqcup \boldsymbol{r}' = \boldsymbol{r}.\boldsymbol{r}' = \boldsymbol{r}'.\boldsymbol{r}; \tag{4.12}$$

that is, the least common extension (least upper bound) of any two consistent firing rules is their concatenation, in either order.

We also need to reconstruct the functional that we used to define the Kahn process. For convenience, let $P_R(\boldsymbol{s})$ denote the set $\{\boldsymbol{r} \in R \mid \boldsymbol{r} \sqsubseteq \boldsymbol{s}\}$. This is a possibly empty finite set. The functional ϕ' is defined such that for any function $F : \mathcal{S}^m \to \mathcal{S}^n$ and any m-tuple $\boldsymbol{s}$,

$$\phi'(F)(\boldsymbol{s}) = \begin{cases} f(\boldsymbol{r}_1).\cdots.f(\boldsymbol{r}_p).F(\boldsymbol{s}') & \text{if } P_R(\boldsymbol{s}) \neq \emptyset \text{ and } \{\boldsymbol{r}_1, \ldots, \boldsymbol{r}_p\} = P_R(\boldsymbol{s}); \\ \boldsymbol{\varepsilon}_n & \text{otherwise.} \end{cases}$$

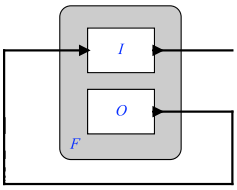

Fig. 4.6. A composition that is invalid under condition (iv), but not under condition (iv′).

Here, we assume, as before, that $\boldsymbol{s} = \boldsymbol{r}_1.\ldots.\boldsymbol{r}_p.\boldsymbol{s}'$. Notice that because of (4.12), for any permutation π on $\{1,\ldots,p\}$,

$$\boldsymbol{r}_1.\ldots.\boldsymbol{r}_p = \boldsymbol{r}_{\pi(1)}.\ldots.\boldsymbol{r}_{\pi(p)},$$

and similarly, because of condition (iv′),

$$f(\boldsymbol{r}_1).\cdots.f(\boldsymbol{r}_p) = f(\boldsymbol{r}_{\pi(1)}).\cdots.f(\boldsymbol{r}_{\pi(p)}).$$

Therefore, it makes no difference in what order we invoke the enabled firing rules. As before, we define the Kahn process F corresponding to the dataflow actor $\langle R, f\rangle$ to be the least fixed point of the functional ϕ'.

Although notationally tedious, it is straightforward to extend the results on ϕ to conclude that both the functional ϕ' and its least fixed point F are continuous; the proofs are practically identical.

Going back to the example of Figure 4.5, we see that we can use the firing rules of (4.10), and a firing function $f : \mathcal{S}^2 \to \mathcal{S}^2$ such that for any firing rule $\boldsymbol{r}$, $f(\boldsymbol{r}) = \boldsymbol{r}$, to obtain a dataflow actor for the two-input two-output identity process that is valid under condition (iv′). More interestingly, we can use the same firing rules to implement a process with firing function $f : \mathcal{S}^2 \to \mathcal{S}$ such that for each firing rule $\boldsymbol{r}$,

$$f(\boldsymbol{r}) = \begin{cases} \langle 1\rangle & \text{if } \boldsymbol{r} = \langle\langle 1\rangle, \langle\ \rangle\rangle \text{ or } \boldsymbol{r} = \langle\langle\ \rangle, \langle 1\rangle\rangle; \\ \langle\ \rangle & \text{otherwise.} \end{cases}$$

This process is interesting because it is neither sequential nor stable, and thus cannot be implemented under condition (iv).

As a final example, consider the composition of Figure 4.6. The top process is an identity process, and the bottom one a source of the infinite sequence $\langle 0, 0, \ldots\rangle$. A reasonable firing function for the source process would be the function $\emptyset \mapsto \langle 0\rangle$. The question now is how to define the firing rules R and firing function f of the composition.

A first, naive attempt would be to let $R = \{\langle 0\rangle, \langle 1\rangle\}$. However, with

the feedback arc in Figure 4.6, this results in no firing rule ever becoming enabled. Instead, we need $R = \{\langle 0\rangle, \langle 1\rangle, \langle\ \rangle\}$, which violates condition (iv). However, if we define the firing function such that

$$f(\langle 0\rangle) = \langle\langle 0\rangle, \langle\ \rangle\rangle,$$
$$f(\langle 1\rangle) = \langle\langle 1\rangle, \langle\ \rangle\rangle, \text{ and}$$
$$f(\langle\ \rangle) = \langle\langle\ \rangle, \langle 0\rangle\rangle,$$

then condition (iv′) is satisfied and the composition behaves as an aggregate of its parts.

4.3.5 Compositionality

The examples of Figure 4.5 and 4.6 indicate certain compositionality issues that can be successfully resolved using the notion of commutative firings. We can generalize this to every composition of the same type.

Consider a slight generalization of Figure 4.1(a), where $\boldsymbol{s}_1$ is an m-tuple, $\boldsymbol{s}_2$ is an n-tuple, $\boldsymbol{s}_3$ is a p-tuple, and $\boldsymbol{s}_4$ is a q-tuple. It is possible to prove that the aggregation of F_1 and F_2 is compositional, in the sense that it can always be described as a set of firing rules and a firing function.

Assume for simplicity that $m > 0$ and $p > 0$ (generalizing to allow zero values is easy), and suppose that F_1 is defined by $\langle R_1, f_1\rangle$ and F_2 by $\langle R_2, f_2\rangle$. Let

$$R_1' = \{\boldsymbol{r}_1 \times \boldsymbol{\varepsilon}_p \mid \boldsymbol{r}_1 \in R_1\}$$

and

$$R_2' = \{\boldsymbol{\varepsilon}_m \times \boldsymbol{r}_2 \mid \boldsymbol{r}_2 \in R_2\},$$

where we loosely write $\boldsymbol{r}_1 \times \boldsymbol{\varepsilon}_p$ to denote the unique $(m+p)$-tuple $\boldsymbol{s}$ that has $\boldsymbol{s}(i) = \boldsymbol{r}_1(i)$ if $i < m$, and $\boldsymbol{s}(i) = \boldsymbol{\varepsilon}_p(i-m)$ otherwise, etc. The set R of firing rules for the composite process $F : \mathcal{S}^{m+p} \to \mathcal{S}^{n+q}$ is defined by

$$R = R_1' \cup R_2'.$$

The firing function $f : \mathcal{S}^{m+p} \to \mathcal{S}^{n+q}$ of the composite process is defined such that for any finite $(m+p)$-tuple $\boldsymbol{r}$,

$$f(\boldsymbol{r}) = \begin{cases} f_1(\boldsymbol{r}_1) \times \boldsymbol{\varepsilon}_q & \text{if } \boldsymbol{r} \in R_1' \text{ and } \boldsymbol{r} = \boldsymbol{r}_1 \times \boldsymbol{\varepsilon}_p; \\ \boldsymbol{\varepsilon}_n \times f_2(\boldsymbol{r}_2) & \text{if } \boldsymbol{r} \in R_2' \text{ and } \boldsymbol{r} = \boldsymbol{\varepsilon}_m \times \boldsymbol{r}_2; \\ \boldsymbol{\varepsilon}_{n+q} & \text{otherwise.} \end{cases}$$

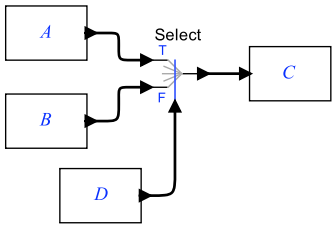

Fig. 4.7. An example of a process network where it might be undesirable from a practical perspective to insist that the operational semantics coincide with the denotational semantics.

It is now straightforward to verify that if $\langle R_1, f_2 \rangle$ and $\langle R_2, f_2 \rangle$ both satisfy condition (iv′), then so does $\langle R, f \rangle$.

4.3.6 Practical Issues

The constructive procedure given by (4.6) ensures that repeated firings converge to the appropriate Kahn process defined by the actor. If any such sequence of firings is finite, then it is only necessary to invoke a finite number of firings. In practice, it is common for such firing sequences to be infinite, in which case a practical issue of fairness arises. In particular, since there are usually many actors in a system, in order to have the operational semantics coincide with the denotational semantics, it is necessary to fire each actor infinitely often, if possible.

It turns out that such a fairness condition is not always desirable. It may result in unbounded memory requirements for execution of a dataflow process network. In some such cases, there is an alternative firing schedule that is also infinite, but requires only bounded memory. That schedule may not conform to the denotational semantics, and nonetheless be preferable to one that does.

A simple example is shown in Figure 4.7. The actor labeled ‘SELECT’ has the following set of firing rules:

$$\{\langle\langle 1\rangle, \langle\ \rangle, \langle 1\rangle\rangle, \langle\langle 0\rangle, \langle\ \rangle, \langle 1\rangle\rangle, \langle\langle\ \rangle, \langle 1\rangle, \langle 0\rangle\rangle, \langle\langle\ \rangle, \langle 0\rangle, \langle 0\rangle\rangle\},$$

where the order of inputs is top-to-bottom. If the bottom input (the control input) has value ‘1’ (for TRUE), then a token of any value is consumed from the top input, and no token is consumed from the middle input. If the control input has value ‘0’ (for FALSE), then a token of any value is consumed from the middle input, and no token is consumed from the top input.

Suppose that the actors A, B, and D, all of which are sources, are defined to each produce an infinite sequence, and that C, which is a sink, is defined to consume an infinite sequence. Suppose further that the output from D is the constant sequence $\langle 0, 0, \ldots \rangle$. Then tokens produced by actor A will never be consumed. In most practical scenarios, it is preferable to avoid producing them if they will never be consumed, despite the fact that this violates the denotational semantics, which state that the output of actor A is an infinite sequence. This problem is solved by Parks [23], who also shows that the obvious solution for the example in Figure 4.7, the demand-driven execution, does not solve the problem in general. Another, more specialized solution, achieved by restricting the semantics, is presented by Caspi in [7].

4.4 Conclusion

We have shown how the formal semantic methods of Kahn dataflow can be adapted to Dennis dataflow, which is based on the notion of an actor firing. Kahn dataflow is defined in terms of continuous processes, which map input sequences to output sequences, while Dennis dataflow is defined in terms of firing functions, which map input tokens to output tokens, and are evaluated only when input tokens satisfy certain firing rules. We have formally defined firing rules and firing functions, and have shown how a Kahn process can be defined as the least fixed point of a continuous functional that is constructed using the firing rules and firing function of an actor. Furthermore, we have specified conditions on the firing rules and firing functions that solve certain compositionality problems in dataflow, in the sense that certain compositions of actors are actors themselves.

Bibliography

[1] Arvind, L. Bic, and T. Ungerer. Evolution of data-flow computers. In J.-L. Gaudiot and L. Bic, editors, *Advanced Topics in Data-Flow Computing.* Prentice-Hall, 1991.

[2] A. Benveniste, P. Caspi, P. L. Guernic, and N. Halbwachs. Data-flow synchronous languages. In J. W. d. Bakker, W.-P. d. Roever, and G. Rozenberg, editors, *A Decade of Concurrency Reflections and Perspectives*, volume 803 of *LNCS*, pages 1–45. Springer-Verlag, Berlin, 1994.

[3] G. Berry. Bottom-up computation of recursive programs. *Revue Franaise dAutomatique, Informatique et Recherche Oprationnelle*, 10(3):47–82, 1976.

[4] J. D. Brock and W. B. Ackerman. Scenarios, a model of non-determinate computation. In *Conference on Formal Definition of Programming Concepts*, volume LNCS 107, pages 252–259. Springer-Verlag, 1981.

[5] M. Broy. Functional specification of time-sensitive communicating systems. *ACM Transactions on Software Engineering and Methodology*, 2(1):1–46, 1993.

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

[7] P. Caspi. Clocks in dataflow languages. *Theoretical Computer Science*, 94(1), 1992.

[8] M. Creeger. Multicore CPUs for the masses. *ACM Queue*, 3(7):63–64, 2005.

[9] B. A. Davey and H. A. Priestly. *Introduction to Lattices and Order*. Cambridge University Press, 1990.

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

[11] M. Geilen and T. Basten. Requirements on the execution of kahn process networks. In *European Symposium on Programming Languages and Systems*, LNCS, pages 319–334. Springer, April 7-11 2003.

[12] C.-J. Hsu, F. Keceli, M.-Y. Ko, S. Shahparnia, and S. S. Bhattacharyya. DIF: An interchange format for dataflow-based design tools. In *International Workshop on Systems, Architectures, Modeling, and Simulation*, Samos, Greece, July 2004.

[13] A. Jantsch and I. Sander. Models of computation and languages for embedded system design. *IEE Proceedings on Computers and Digital Techniques*, 152(2):114–129, 2005.

[14] W. M. Johnston, J. R. P. Hanna, and R. J. Millar. Advances in dataflow programming languages. *ACM Computing Surveys*, 36(1):1–34, 2004.

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

[16] G. Kahn and D. B. MacQueen. Coroutines and networks of parallel processes. In B. Gilchrist, editor, *Information Processing*, pages 993–998. North-Holland Publishing Co., 1977.

[17] D. Lazaro Cuadrado, A. P. Ravn, and P. Koch. Automated distributed simulation in Ptolemy II. In *Parallel and Distributed Computing and Networks (PDCN)*. Acta Press, February 13 15 2007.

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

[19] Y. Lin, R. Mullenix, M. Woh, S. Mahlke, T. Mudge, A. Reid, and K. Flautner. SPEX: A programming language for software defined radio. In *Software Defined Radio Technical Conference and Product Exposition*, Orlando, November 13-17 2006.

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

[21] A. G. Olson and B. L. Evans. Deadlock detection for distributed process networks. In *ICASSP*, 2005.

[22] O. M. G. (OMG). A UML profile for MARTE, beta 1. OMG Adopted

Specification ptc/07-08-04, August 2007.

[23] T. M. Parks. *Bounded Scheduling of Process Networks*. Phd, UC Berkeley, 1995.

[24] T. M. Parks and D. Roberts. Distributed process networks in Java. In *International Parallel and Distributed Processing Symposium*, Nice, France, April 2003.

[25] D. Scott. Outline of a mathematical theory of computation. In *4th annual Princeton conf. on Information sciences and systems*, pages 169–176, 1970.

[26] V. Srini. An architectural comparison of dataflow systems. *Computer*, 19(3), 1986.

[27] E. W. Stark. An algebra of dataflow networks. *Fundamenta Informaticae*, 22(1-2):167–185, 1995.

[28] R. Stephens. A survey of stream processing. *Acta Informatica*, 34(7), 1997.

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

[30] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A language for streaming applications. In *11th International Conference on Compiler Construction*, volume LNCS 2304, Grenoble, France, April 8-12, 2002 2002. Springer-Verlag.

[31] W. Thies, M. Karczmarek, J. Sermulins, R. Rabbah, and S. Amarasinghe. Teleport messaging for distributed stream programs. In *PPoPP*, Chicago, Illinois, USA, June 1517 2005. ACM.

[32] A. Turjan, B. Kienhuis, and E. Deprettere. Solving out-of-order communication in Kahn process networks. *Journal on VLSI Signal Processing-Systems for Signal, Image, and Video Technology*, 2003.