A Calculus of Dataflow Networks

(Extended Abstract)

Eugene W. Stark*
Department of Computer Science
State University of New York at Stony Brook
Stony Brook, NY 11794 USA†

Abstract

Dataflow networks are a paradigm for concurrent computation in which a collection of concurrently and asynchronously executing processes communicate by sending data values over FIFO communication channels. In this paper, we define a CCS-style calculus of dataflow networks with a standard structural operational semantics. A version of weak bisimulation equivalence, called "buffer bisimilarity," is defined for this calculus, and its equational theory is investigated. The main result is a completeness theorem for proving equations valid under buffer bisimilarity. The axioms have a familiar, category-theoretic flavor, in which a dataflow process with m input ports and n output ports is represented by an arrow from m to n in a category whose objects are the finite ordinals.

1 Introduction

Dataflow networks, originally introduced by Kahn [4], are a paradigm for concurrent computation in which a collection of concurrently and asynchronously executing processes communicate by sending data values over FIFO communication channels. In his original paper, and in a subsequent paper with MacQueen [5], Kahn considered *determinate* networks, in which processes are expressed in a programming language restricted in such a way that processes compute functions, from the history of values arriving on their input ports to the history of values emitted at their output ports. Kahn made the insightful observation that networks of such processes also compute functions, and that the function computed by a network is related to the functions computed by their component processes by an elegant least-fixed-point principle.

Although dataflow networks were one of the first paradigms for concurrent computation to be studied, recent work on them has diverged significantly from mainstream efforts in concurrency theory, for example those centered around CCS [8] and ACP [1]. Studies of dataflow networks, including my own, have tended to eschew concrete syntax in favor of a more abstract semantic approach, and this has often made the results of these studies hard to interpret in terms of concrete computational intuition. Perhaps one reason for the semantic approach is that much of the research on dataflow networks has been motivated by the observation, due to Brock and Ackerman [2], that Kahn's fixed-point principle does not extend in a naive way to *indeterminate* processes, which are capable of making unpredictable choices during a computation. Since Kahn's principle is a direct application of ideas from denotational semantics, it has perhaps seemed natural that investigations of how to extend this principle to indeterminate networks should be performed in a similar abstract semantic context. In my own studies of these issues [7, 11, 12, 14], I have consciously avoided committing to a particular syntax until I was confident that the syntax would clarify the underlying mathematical structure, rather than obscuring it amid a mass of irrelevant detail. I now feel confident enough to propose a particular syntactic formulation.

The purpose of this paper is therefore to take steps to bring the theory of dataflow networks back into the mainstream of work on concurrency theory, and in doing so, to show how my previous work fits into the mainstream. This will be accomplished by defining a CCS-style calculus for dataflow networks, complete with a structural operational semantics [10] and an equivalence based on bisimulation [9], and by giving a sound and complete axiomatization of this equivalence. A novel feature of the calculus is the way in which "buffers" are incorporated into each of the primitive processes and operations, to permit a smooth handling of asynchronous communication. The formulation has a decidedly category-theoretic flavor, in which we think of a dataflow process or network f with m input ports and n output ports as an "arrow" $f : m \to n$.

*Research supported in part by NSF Grant CCR-8902215.
†E-mail address: `stark@cs.sunysb.edu` (Internet)

Similarly, a transition in which a process f performs some computational step and becomes process g is thought of as an "arrow of processes" $f \Longrightarrow g$. We are concerned here with several types of operations for forming new processes from old ones: *input* and *output buffering*, in which a buffer for data values is attached either to the input or output of a process, *sequential composition*, in which the output ports of a process $f : m \to p$ are connected to the corresponding input ports of a process $g : p \to n$, resulting in a process $f;g : m \to n$, *parallel composition*, in which a process $f : m \to n$ is juxtaposed with a process $f' : m' \to n'$ to yield a process $f \otimes f' : m + m' \to n + n'$, and *feedback*, in which the "rightmost p output ports" of a process $f : m + p \to n + p$ are connected to the corresponding "rightmost p input ports" of the same process, resulting in a process $f \circlearrowleft_p : m \to n$. These process-forming operations can be thought of as axioms and inference rules for inferring processes. Similarly, the structural operational semantics can be thought of as a collection of axioms and rules for inferring transitions. This point of view is similar to that applied to the λ-calculus in [6]. The category-theoretic flavor extends to the equational axioms, a number of which are familiar category-theoretic properties.

2 Dataflow Calculus

In this section, we define a system that formalizes the intuitive conception of dataflow networks as collections of concurrent processes communicating by passing data values over FIFO channels. The formal system, called *dataflow calculus*, comprises three different kinds of entities:

1. For $n \in \mathrm{Ord}$ (the set of finite ordinals) a monoid $\mathrm{Buf}(n)$ of *buffers of width* n. These are defined below.

2. A set Proc of *processes*, equipped with functions $\mathrm{iwd} : \mathrm{Proc} \to \mathrm{Ord}$ and $\mathrm{owd} : \mathrm{Proc} \to \mathrm{Ord}$ that map each process to its *input width* and *output width*, respectively. We write $\mathrm{Proc}(m,n)$ to denote the set of processes with input width m and output width n.

3. A set Trans of *transitions*, equipped with functions $\mathrm{src} : \mathrm{Trans} \to \mathrm{Proc}$ and $\mathrm{trg} : \mathrm{Trans} \to \mathrm{Proc}$ that map each transition to its *source* and *target* processes, respectively.

A *buffer* is a function $\beta : [n] \to \mathcal{N}^*$, where $[n]$ is the set $\{0, 1, 2, \ldots, n-1\}$, and $\mathcal{N}^*$ is the monoid of finite strings of natural numbers. The number n is called the *width* of the buffer β, and we use $\mathrm{Buf}(n)$ to denote the set of all buffers of width n. (Intuitively, one should think of one of our buffers of width n as consisting of n ordinary FIFO buffers placed side-by-side.) We denote particular elements of $\mathrm{Buf}(n)$ by expresssions of the form

$$\langle 0 : s_0, 1 : s_1, \ldots, (n-1) : s_{n-1} \rangle$$

where s_0, s_1, ..., s_{n-1} are elements of $\mathcal{N}^*$. Such an expression stands for the buffer $\beta \in \mathrm{Buf}(n)$ whose value on i is s_i. When the width of the buffer is clear from the context, we feel free to omit some of the $i : s$ pairs, with the understanding that we mean the omitted pair to be $i : \epsilon$, where ϵ is the empty string. We also regard $\mathrm{Buf}(n)$ as included in $\mathrm{Buf}(n')$ for all $n' \geq n$, under the natural "extend with ϵ" map. Buffers of the same width can be concatenated componentwise, making each set $\mathrm{Buf}(n)$ into a monoid. We denote buffer concatenation simply by juxtaposition. Besides this "vertical composition" of buffers, there is also a "horizontal composition," in which a buffer of width m is placed next to a buffer of width n to obtain a buffer of width $m+n$. We denote this kind of composition (which is clearly associative, and has the buffer $\langle\rangle$ of width 0 as a unit) by $\otimes$.

Processes are terms that are built up from certain *basic standard processes* and *basic nonstandard processes*, using certain *process-forming operations*, all of which are described in more detail below. We write $f : m \to n$ to assert that f is a process with $\mathrm{iwd}(f) = m$ and $\mathrm{owd}(f) = n$.

A *transition* is an expression Γ of one of the three forms: $f \overset{i?v}{\Longrightarrow} g$, $f \overset{j!v}{\Longrightarrow} g$, or $f \overset{k\#v}{\Longrightarrow} g$, where f and g are processes with $\mathrm{iwd}(f) = \mathrm{iwd}(g) = m$ and $\mathrm{owd}(f) = \mathrm{owd}(g) = n$, the values i, j, and k are elements of $[m]$, $[n]$, and Ord, respectively, and $v \in \mathcal{N}$ is a data value. In each case, $\mathrm{src}(\Gamma) = f$ and $\mathrm{trg}(\Gamma) = g$. A transition is called an *input transition*, *output transition*, or an *internal transition*, respectively, according to which of the three forms it has. We call the particular designation (input, output, or internal) assigned to a transition its *mode*, we call the buffer $\langle i : v \rangle$, $\langle j : v \rangle$, or $\langle k : v \rangle$ associated, respectively, with a transition the *token* associated with the transition, and we call the ordinal i, j, or k the *port* of the transition. We write simply $\Gamma : f \overset{?}{\Longrightarrow} g$, $\Gamma : f \overset{!}{\Longrightarrow} g$, or $\Gamma : f \overset{\#}{\Longrightarrow} g$ when we are interested only in the mode of a transition and not in its token. When we are not even interested in the mode of the transition Γ, we merely write $\Gamma : f \Longrightarrow g$. Transitions are inferred from *transition axioms* using *transition inference rules*, as detailed below. The transition axioms and inference rules amount to an SOS (Structural Operational Semantics) definition of the computational behavior of processes.

Basic Standard Processes

Dataflow calculus contains the following basic standard processes for all $n \in \mathrm{Ord}$ and $\beta, \beta' \in \mathrm{Buf}(n)$:

- An *identity process* $\mathbf{1}_n\{\delta\} : n \to n$ for all $n \in \mathrm{Ord}$ and all $\delta \in \mathrm{Buf}(n)$.
- A *terminator process* $\mathbf{t}_n : n \to 0$ for all $n \in \mathrm{Ord}$.
- A *generator process* $\mathbf{g}_n\{\delta\} : 0 \to n$ for all $n \in \mathrm{Ord}$ and all $\delta \in \mathrm{Buf}(n)$.
- A *duplicator process* $\mathbf{d}_n\{\delta, \delta'\} : n \to n + n$ for all $n \in \mathrm{Ord}$ and all $\delta, \delta' \in \mathrm{Buf}(n)$.
- An *exchange process* $\mathbf{s}_{m,n}\{\eta, \delta\} : m + n \to n + m$ for all $m, n \in \mathrm{Ord}$, all $\delta \in \mathrm{Buf}(m)$ and all $\eta \in \mathrm{Buf}(n)$.

These processes are used to perform the "wiring" in dataflow networks, and, in the case of the generator processes, to serve as a source of data values. The example process in the next section should serve to clarify these issues.

Basic Nonstandard Processes

A particular dataflow calculus may have an arbitrary set (possibly empty) of basic nonstandard processes, about which we assume nothing other than that each has a specified input and output width.

Process-Forming Operations

Dataflow calculus has the following process-forming operations:

- If $f : m \to n$ and $\beta \in \mathrm{Buf}(m)$, then there is a process $\beta f : m \to n$, called the *input buffering* of f by β.
- If $f : m \to n$ and $\gamma \in \mathrm{Buf}(n)$, then there is a process $f\gamma : m \to n$, called the *output buffering* of f by γ.
- If $f : m \to p$, $g : p \to n$, and $\delta \in \mathrm{Buf}(p)$ then there is a process $f\{\delta\}g : m \to n$, called the *buffered sequential composition* of f and g with δ.
- If $f : m \to n$ and $f' : m' \to n'$, then there is a process $f \otimes f' : m+m' \to n+n'$, called the *parallel composition* of f and f'.
- If $f : m + p \to n + p$, and $\delta \in \mathrm{Buf}(p)$ then there is a process $f \circlearrowleft_p \{\delta\} : m \to n$, called the *buffered feedback* of f with δ.

We assume there are no processes other than the ones that can be built from basic standard processes and basic nonstandard processes, using the process-forming operations.

$$
\begin{array}{rcl}
\mathbf{1}_n\{\delta\} & \overset{j?v}{\Longrightarrow} & \mathbf{1}_n\{\langle j : v\rangle\delta\} \\
\mathbf{1}_n\{\delta\langle j : v\rangle\} & \overset{j!v}{\Longrightarrow} & \mathbf{1}_n\{\delta\} \\
\mathbf{t}_n & \overset{j?v}{\Longrightarrow} & \mathbf{t}_n \\
\mathbf{g}_n\{\delta\langle j : v\rangle\} & \overset{j!v}{\Longrightarrow} & \mathbf{g}_n\{\langle j : v\rangle\delta\} \\
\mathbf{d}_n\{\delta, \delta'\} & \overset{j?v}{\Longrightarrow} & \mathbf{d}_n\{\langle j : v\rangle\delta, \langle j : v\rangle\delta'\} \\
\mathbf{d}_n\{\delta\langle j : v\rangle, \delta'\} & \overset{j!v}{\Longrightarrow} & \mathbf{d}_n\{\delta, \delta'\} \\
\mathbf{d}_n\{\delta, \delta'\langle j : v\rangle\} & \overset{(j+n)!v}{\Longrightarrow} & \mathbf{d}_n\{\delta, \delta'\} \\
\mathbf{s}_{m,n}\{\eta, \delta\} & \overset{i?v}{\Longrightarrow} & \mathbf{s}_{m,n}\{\eta, \langle i : v\rangle\delta\} \\
\mathbf{s}_{m,n}\{\eta, \delta\} & \overset{(j+m)?v}{\Longrightarrow} & \mathbf{s}_{m,n}\{\langle j : v\rangle\eta, \delta\} \\
\mathbf{s}_{m,n}\{\eta, \delta\langle i : v\rangle\} & \overset{(i+n)!v}{\Longrightarrow} & \mathbf{s}_{m,n}\{\eta, \delta\} \\
\mathbf{s}_{m,n}\{\eta\langle j : v\rangle, \delta\} & \overset{j!v}{\Longrightarrow} & \mathbf{s}_{m,n}\{\eta, \delta\},
\end{array}
$$

Figure 1: Transition Axioms for Basic Standard Processes

Since process expressions can be hard to read, it is convenient to introduce some simplifying abbreviations. Specifically,

- $\mathbf{1}_n$, $\mathbf{0}_n$, $\mathbf{d}_n$, and $\mathbf{s}_{m,n}$ abbreviate $\mathbf{1}_n\{\langle\rangle\}$, $\mathbf{g}_n\{\langle\rangle\}$, $\mathbf{d}_n\{\langle\rangle, \langle\rangle\}$, and $\mathbf{s}_{m,n}\{\langle\rangle, \langle\rangle\}$, respectively.
- $f; g$ abbreviates $f\{\langle\rangle\}g$.
- $f\circlearrowleft_p$ abbreviates $f\circlearrowleft_p \{\langle\rangle\}$.

As a consequence of the semantics we give below for $\mathbf{g}_n\{\langle\rangle\}$, the process $\mathbf{0}_n$, which has no input ports and n output ports, is a process that never emits any output data whatsoever.

Transition Axioms and Inference Rules

Dataflow calculus contains the transition axioms for the basic standard processes shown in Figure 1, where in each case we assume that $i \in [m]$, $j \in [n]$, and that the buffers all have appropriate widths. In addition, each of the process-forming operations has an associated collection of transition axioms and inference rules. These are listed in Figure 2.

A particular dataflow calculus may also contain additional transition axioms, but no other inference rules. We call the additional axioms *nonstandard transition axioms*, and we require that the set of all such axioms satisfy the following conditions:

Format: The source and target of each nonstandard transition axiom must both be basic nonstandard processes.

Disambiguation: Suppose $\Gamma : f \Longrightarrow g$ and $\Gamma' : f \Longrightarrow g'$ are nonstandard transition axioms having the same mode and token. Then $g = g'$.

Input Buffering Axiom and Rules:

$$\beta f \quad \overset{i?v}{\Longrightarrow} \quad (\langle i : v\rangle\beta)f \qquad \frac{f \overset{i?v}{\Longrightarrow} f'}{(\beta\langle i : v\rangle)f \overset{i\#v}{\Longrightarrow} \beta f'} \qquad \frac{f \overset{k\#v}{\Longrightarrow} f'}{\beta f \overset{(k+m)\#v}{\Longrightarrow} \beta f'} \qquad \frac{f \overset{i!v}{\Longrightarrow} f'}{\beta f \overset{i!v}{\Longrightarrow} \beta f'}$$

Output Buffering Axiom and Rules:

$$\frac{f \overset{i?v}{\Longrightarrow} f'}{f\gamma \overset{i?v}{\Longrightarrow} f'\gamma} \qquad \frac{f \overset{k\#v}{\Longrightarrow} f'}{f\gamma \overset{(k+n)\#v}{\Longrightarrow} f'\gamma} \qquad \frac{f \overset{j!v}{\Longrightarrow} f'}{f\gamma \overset{j\#v}{\Longrightarrow} f'(\langle j : v\rangle\gamma)} \qquad f(\gamma\langle j : v\rangle) \quad \overset{j!v}{\Longrightarrow} \quad f\gamma$$

Sequential Composition Rules:

$$\frac{f \overset{i?v}{\Longrightarrow} f'}{f\{\delta\}g \overset{i?v}{\Longrightarrow} f'\{\delta\}g} \qquad \frac{f \overset{k\#v}{\Longrightarrow} f'}{f\{\delta\}g \overset{(2k+2p)\#v}{\Longrightarrow} f'\{\delta\}g} \qquad \frac{f \overset{k!v}{\Longrightarrow} f'}{f\{\delta\}g \overset{(k+p)\#v}{\Longrightarrow} f'\{\langle k : v\rangle\delta\}g}$$

$$\frac{g \overset{j!v}{\Longrightarrow} g'}{f\{\delta\}g \overset{j!v}{\Longrightarrow} f\{\delta\}g'} \qquad \frac{g \overset{k\#v}{\Longrightarrow} g'}{f\{\delta\}g \overset{(2k+2p+1)\#v}{\Longrightarrow} f\{\delta\}g'} \qquad \frac{g \overset{k?v}{\Longrightarrow} g'}{f\{\delta\langle k : v\rangle\}g \overset{k\#v}{\Longrightarrow} f\{\delta\}g'}$$

Parallel Composition Rules:

$$\frac{f \overset{i?v}{\Longrightarrow} g}{f \otimes f' \overset{i?v}{\Longrightarrow} g \otimes f'} \qquad \frac{f \overset{k\#v}{\Longrightarrow} g}{f \otimes f' \overset{(2k)\#v}{\Longrightarrow} g \otimes f'} \qquad \frac{f \overset{j!v}{\Longrightarrow} g}{f \otimes f' \overset{j!v}{\Longrightarrow} g \otimes f'}$$

$$\frac{f' \overset{i?v}{\Longrightarrow} g'}{f \otimes f' \overset{(i+m)?v}{\Longrightarrow} f \otimes g'} \qquad \frac{f' \overset{k\#v}{\Longrightarrow} g'}{f \otimes f' \overset{(2k+1)\#v}{\Longrightarrow} f \otimes g'} \qquad \frac{f' \overset{j!v}{\Longrightarrow} g'}{f \otimes f' \overset{(j+n)!v}{\Longrightarrow} f \otimes g'}$$

Feedback Rules:

$$\frac{f \overset{i?v}{\Longrightarrow} g \quad i < m}{f\circlearrowleft_p \{\delta\} \overset{i?v}{\Longrightarrow} g\circlearrowleft_p \{\delta\}} \qquad \frac{f \overset{k\#v}{\Longrightarrow} g}{f\circlearrowleft_p \{\delta\} \overset{(k+2p)\#v}{\Longrightarrow} g\circlearrowleft_p \{\delta\}} \qquad \frac{f \overset{j!v}{\Longrightarrow} g \quad j < n}{f\circlearrowleft_p \{\delta\} \overset{j!v}{\Longrightarrow} g\circlearrowleft_p \{\delta\}}$$

$$\frac{f \overset{(k+m)?v}{\Longrightarrow} g \quad k < p}{f\circlearrowleft_p \{\delta\langle k : v\rangle\} \overset{k\#v}{\Longrightarrow} g\circlearrowleft_p \{\delta\}} \qquad \frac{f \overset{(k+n)!v}{\Longrightarrow} g \quad k < p}{f\circlearrowleft_p \{\delta\} \overset{(k+p)\#v}{\Longrightarrow} g\circlearrowleft_p \{\langle k : v\rangle\delta\}}$$

Figure 2: Transition Inference Rules

A transition is called *standard* if can be inferred without making use of any nonstandard transition axioms.

The purpose of the Disambiguation condition, combined with the way in which the transition rules of Figure 2 assign ports to internal transitions, is to make sure that each transition in a dataflow calculus is uniquely determined by its source process, its mode, and its token. In addition, the tokens of internal transitions are used to define a syntactic notion of "independence" of transitions (see Section 6). We can do away with the Disambiguation condition, as well as the annotations of internal transitions, if we are willing to systematically introduce the notion of "proof terms" for transitions, so that each transition is uniquely determined by its proof term, and so that independence is determined by comparing proof terms. We do not follow this somewhat more complicated approach in this paper.

It will be useful to have names for various classes of processes in a dataflow calculus. A process is called *standard* if it contains no basic nonstandard processes. A process is *feedback-free* if it contains no occurrences of a feedback operation. A process is called *reticulate* if it is a standard process built up from processes $\mathbf{1}\{\langle\rangle\}$, $\mathbf{0}$, $\mathbf{t}$, $\mathbf{s}\{\langle\rangle,\langle\rangle\}$, and $\mathbf{d}\{\langle\rangle\langle\rangle\}$, using any of the process-forming operations but feedback. A *pure reticulate* process is one that is built up from $\mathbf{1}\{\langle\rangle\}$, $\mathbf{0}$, $\mathbf{t}$, $\mathbf{s}\{\langle\rangle,\langle\rangle\}$, and $\mathbf{d}\{\langle\rangle,\langle\rangle\}$, using ; and $\otimes$ only.

Dataflow calculi of particular interest to us are the dataflow calculi $\mathcal{D}(X)$ determined by taking a given set $X = \{f, g, h, \ldots\}$, equipped with functions iwd : $X \rightarrow$ Ord and owd : $X \rightarrow$ Ord, as the basic nonstandard processes, and assuming no transition axioms other than the standard ones mentioned above. We call $\mathcal{D}(X)$ the *generic* dataflow calculus associated with the set X.

3 Example

Consider the process

$$(((\mathbf{0}_1\langle 0:5\rangle)\otimes\mathbf{1}_1);((\mathbf{m}\{\langle\rangle\}\langle 0:7\rangle);\mathbf{d}_1))\circlearrowleft_1$$

in the dataflow calculus whose basic nonstandard processes are all $\mathbf{m}\{\delta\}: 2 \rightarrow 1$, for $\delta \in \mathrm{Buf}(1)$. This process may be visualized as the dataflow network shown in Figure 3, in which processes are represented as polygons, values in buffers as circles, and interconnections as arrows.

Assume the following set of nonstandard transition axioms:

$$\mathbf{m}\{\delta\} \overset{0?v}{\Longrightarrow} \mathbf{m}\{\langle 0:v\rangle\delta\}$$
$$\mathbf{m}\{\delta\} \overset{1?v}{\Longrightarrow} \mathbf{m}\{\langle 0:v\rangle\delta\}$$
$$\mathbf{m}\{\delta\langle 0:v\rangle\} \overset{0!v}{\Longrightarrow} \mathbf{m}\{\delta\}.$$

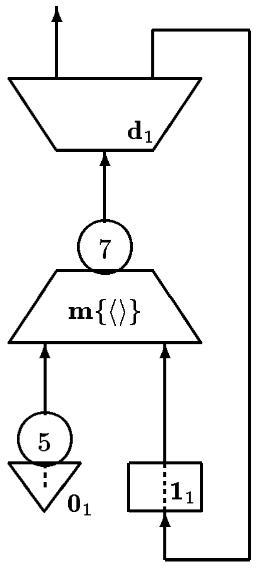

Figure 3: Example Dataflow Network

These axioms imply that $\mathbf{m}\{\langle\rangle\}$ behaves as a "merge" process, which takes the sequences of values arriving on its two input ports, and shuffles them together in an indeterminate order to produce a single output sequence.

From the nonstandard transition axioms, plus the other axioms and rules for dataflow calculus, we can infer the transition sequence shown in Figure 4. In this computation, the data value 5 first passes through the merge process and then the value 7 passes the duplicator, followed by the value 5. One copy of the value 7 is output, and the other copy makes its way around the feedback loop, eventually passing through the merge process and duplicator. In the last step, one of the copies of the value 5 is output. By tracing through this transition sequence, one can see how in dataflow calculus, the standard processes are used for "wiring," and how the calculus models the flow of data values through a network.

4 Process Equivalence

If $f : m \rightarrow n$ and $g : m \rightarrow n$ are processes in a dataflow calculus $\mathcal{C}$, then an *internal computation from f to g* is a sequence of k transitions (where $k \geq 0$):

$$h_0 \overset{\#}{\Longrightarrow} h_1 \overset{\#}{\Longrightarrow} \ldots \overset{\#}{\Longrightarrow} h_k,$$

with $h_0 = f$ and $h_k = g$. We write $f \overset{\#^*}{\Longrightarrow} g$ to assert the existence of an internal computation from f to g.

A *simulation* on $\mathrm{Proc}(m,n)$ is a binary relation $\mathcal{R}$ on $\mathrm{Proc}(m,n)$ such that whenever $f\ \mathcal{R}\ f'$, then the following three conditions hold:

$$\begin{aligned}
&(((\mathbf{0}_1\langle 0:5\rangle)\otimes \mathbf{1}_1);((\mathbf{m}\{\langle\rangle\}\langle 0:7\rangle);\mathbf{d}_1))\circlearrowleft_1\\
&\quad\overset{\#}{\Longrightarrow}\quad ((\mathbf{0}_1\langle\rangle\otimes \mathbf{1}_1)\{\langle 0:5\rangle\}((\mathbf{m}\{\langle\rangle\}\langle 0:7\rangle);\mathbf{d}_1))\circlearrowleft_1\\
&\quad\overset{\#}{\Longrightarrow}\quad ((\mathbf{0}_1\langle\rangle\otimes \mathbf{1}_1);((\mathbf{m}\{\langle 0:5\rangle\}\langle 0:7\rangle);\mathbf{d}_1))\circlearrowleft_1\\
&\quad\overset{\#}{\Longrightarrow}\quad ((\mathbf{0}_1\langle\rangle\otimes \mathbf{1}_1);((\mathbf{m}\{\langle\rangle\}\langle 0:57\rangle);\mathbf{d}_1))\circlearrowleft_1\\
&\quad\overset{\#}{\Longrightarrow}\quad ((\mathbf{0}_1\langle\rangle\otimes \mathbf{1}_1);((\mathbf{m}\{\langle\rangle\}\langle 0:5\rangle)\{\langle 0:7\rangle\}\mathbf{d}_1))\circlearrowleft_1\\
&\quad\overset{\#}{\Longrightarrow}\quad ((\mathbf{0}_1\langle\rangle\otimes \mathbf{1}_1);((\mathbf{m}\{\langle\rangle\}\langle\rangle)\{\langle 0:57\rangle\}\mathbf{d}_1))\circlearrowleft_1\\
&\quad\overset{\#}{\Longrightarrow}\quad ((\mathbf{0}_1\langle\rangle\otimes \mathbf{1}_1);((\mathbf{m}\{\langle\rangle\}\langle\rangle)\{\langle 0:5\rangle\}\mathbf{d}_1\{\langle 0:7\rangle,\langle 0:7\rangle\}))\circlearrowleft_1\\
&\quad\overset{\#}{\Longrightarrow}\quad ((\mathbf{0}_1\langle\rangle\otimes \mathbf{1}_1);((\mathbf{m}\{\langle\rangle\}\langle\rangle);\mathbf{d}_1\{\langle 0:57\rangle,\langle 0:57\rangle\}))\circlearrowleft_1\\
&\quad\overset{0!7}{\Longrightarrow}\quad ((\mathbf{0}_1\langle\rangle\otimes \mathbf{1}_1);((\mathbf{m}\{\langle\rangle\}\langle\rangle);\mathbf{d}_1\{\langle 0:5\rangle,\langle 0:57\rangle\}))\circlearrowleft_1\\
&\quad\overset{\#}{\Longrightarrow}\quad ((\mathbf{0}_1\langle\rangle\otimes \mathbf{1}_1);((\mathbf{m}\{\langle\rangle\}\langle\rangle);\mathbf{d}_1\{\langle 0:5\rangle,\langle 0:5\rangle\}))\circlearrowleft_1\ \{\langle 0:7\rangle\}\\
&\quad\overset{\#}{\Longrightarrow}\quad ((\mathbf{0}_1\langle\rangle\otimes \mathbf{1}_1\{\langle 0:7\rangle\});((\mathbf{m}\{\langle\rangle\}\langle\rangle);\mathbf{d}_1\{\langle 0:5\rangle,\langle 0:5\rangle\}))\circlearrowleft_1\\
&\quad\overset{\#}{\Longrightarrow}\quad ((\mathbf{0}_1\langle\rangle\otimes \mathbf{1}_1)\{\langle 0:7\rangle\}((\mathbf{m}\{\langle\rangle\}\langle\rangle);\mathbf{d}_1\{\langle 0:5\rangle,\langle 0:5\rangle\}))\circlearrowleft_1\\
&\quad\overset{\#}{\Longrightarrow}\quad ((\mathbf{0}_1\langle\rangle\otimes \mathbf{1}_1);((\mathbf{m}\{\langle 0:7\rangle\}\langle\rangle);\mathbf{d}_1\{\langle 0:5\rangle,\langle 0:5\rangle\}))\circlearrowleft_1\\
&\quad\overset{\#}{\Longrightarrow}\quad ((\mathbf{0}_1\langle\rangle\otimes \mathbf{1}_1);((\mathbf{m}\{\langle\rangle\}\langle 0:7\rangle);\mathbf{d}_1\{\langle 0:5\rangle,\langle 0:5\rangle\}))\circlearrowleft_1\\
&\quad\overset{\#}{\Longrightarrow}\quad ((\mathbf{0}_1\langle\rangle\otimes \mathbf{1}_1);((\mathbf{m}\{\langle\rangle\}\langle\rangle)\{\langle 0:7\rangle\}\mathbf{d}_1\{\langle 0:5\rangle,\langle 0:5\rangle\}))\circlearrowleft_1\\
&\quad\overset{\#}{\Longrightarrow}\quad ((\mathbf{0}_1\langle\rangle\otimes \mathbf{1}_1);((\mathbf{m}\{\langle\rangle\}\langle\rangle);\mathbf{d}_1\{\langle 0:75\rangle,\langle 0:75\rangle\}))\circlearrowleft_1\\
&\quad\overset{0!5}{\Longrightarrow}\quad ((\mathbf{0}_1\langle\rangle\otimes \mathbf{1}_1);((\mathbf{m}\{\langle\rangle\}\langle\rangle);\mathbf{d}_1\{\langle 0:7\rangle,\langle 0:75\rangle\}))\circlearrowleft_1.
\end{aligned}$$

Figure 4: Transition Sequence for Example Process

1. If $f\overset{i?v}{\Longrightarrow}g$, then there exists a computation of the following form:
$$f'\overset{\#^*}{\Longrightarrow}x\overset{i?v}{\Longrightarrow}y\overset{\#^*}{\Longrightarrow}g'$$
such that $g\,\mathcal{R}\,g'$.

2. If $f\overset{j!v}{\Longrightarrow}g$, then there exists a computation of the following form:
$$f'\overset{\#^*}{\Longrightarrow}x\overset{j!v}{\Longrightarrow}y\overset{\#^*}{\Longrightarrow}g'$$
such that $g\,\mathcal{R}\,g'$.

3. If $f\overset{\#}{\Longrightarrow}g$, then there exists a computation of the following form:
$$f'\overset{\#^*}{\Longrightarrow}g'$$
such that $g\,\mathcal{R}\,g'$.

A simulation whose converse is also a simulation is called a *bisimulation.* We say that f and g are *bisimilar*, and we write $f\approx g$, if there exists a bisimulation $\mathcal{R}$ such that $f\,\mathcal{R}\,g$. It can be shown that this definition of bisimulation has the usual properties, including the fact that the bisimilarity relation is itself a bisimulation, and that bisimilarity is a congruence with respect to the process-forming operations. It can also be shown that if $f\overset{\#}{\Longrightarrow}g$ is a standard internal transition, then $f\approx g$.

Although bisimilarity is an interesting equivalence, we do *not* in general have $\langle\rangle f\approx f$ or $f\langle\rangle\approx f$, if we allow arbitrary sets of transition axioms for nonstandard processes. Intuitively, the reason is that f might not have any computation to correspond to an internal transition $f\langle\rangle\overset{\#}{\Longrightarrow}f'\langle j:v\rangle$, or to an input transition $\langle\rangle f\overset{i?v}{\Longrightarrow}\langle i:v\rangle f$. Since we wish to have an equivalence that places all of f, $\langle\rangle f$, and $f\langle\rangle$ in the same equivalence class, we modify bisimilarity somewhat. Formally, define the *buffer bisimilarity* relation $\overset{\mathrm{b}}{\approx}$ by:

$$f\overset{\mathrm{b}}{\approx}g\ \text{ iff }\ \langle\rangle(f\langle\rangle)\approx\langle\rangle(g\langle\rangle).$$

It follows from the fact that bisimilarity is a congruence that buffer bisimilarity is a coarser relation than bisimilarity. Moreover, it can be shown that buffer bisimilarity is also a congruence, and that $\langle\rangle f\overset{\mathrm{b}}{\approx}f$ and $f\overset{\mathrm{b}}{\approx}f\langle\rangle$ hold.

5 Equational Laws

Our main result is a proof-theoretic characterization of the equational theory of buffer bisimilarity. To state the result precisely, we have to define what it means for an equation with variables to be "valid" in dataflow calculus. Formally, define an *equation with variables in* X to be a pair (f, g) of processes in the generic dataflow calculus $\mathcal{D}(X)$. Usually, we denote an equation by the expression $f = g$ instead of the tuple notation (f, g). If $\mathcal{C}$ is any dataflow calculus, then an *interpretation* of X in $\mathcal{C}$ is a function $I : X \to \mathcal{C}$ that preserves input and output width. Such a function extends inductively to a mapping from all of $\mathcal{D}(X)$ to $\mathcal{C}$. We say that an equation $f = g$ with variables in X is *true* under an interpretation $I : \mathcal{D}(X) \to \mathcal{C}$, and we write $\models_I f = g$, if $I(f) \overset{\mathrm{b}}{\approx} I(g)$ holds in $\mathcal{C}$. We say that $f = g$ is *valid*, and we write $\models f = g$, if $\models_I f = g$ holds for all interpretations.

We can now give a long list of valid equations. Figures 5–8 list four sets of equational laws:

- *extraction laws.*
- *categorical laws.*
- *buffering laws.*
- *feedback laws.*

In each case, we assume that the widths of the processes and buffers involved are such that both sides of the equation make sense and have the same input and output widths. The laws are not all independent: (Zer1)-(Zer3) are provable using the feedback laws, and in addition it is conceivable that the feedback laws are not all independent.

The extraction laws in Figure 5 show how we can extract nonempty buffers from within basic standard processes and from sequential composition and feedback. These laws show that for the purposes of studying the equational theory most of the time no generality is lost by using only basic standard processes with empty buffers, and by working with $f; g$ and $f \circlearrowleft_p$, rather than the more general versions. The only exception is the generator process $\mathbf{g}_n\{\delta\}$, from which a nonempty buffer δ cannot be extracted. However, even though buffers are almost superfluous for the equational theory, they are absolutely essential to obtaining a simple operational semantics, and this is why we cannot do away with them entirely.

Many of the laws in Figures 6 and 7 are familiar. The laws (Cat1)-(Cat3) are the axioms for a category. The laws (Trm1)-(Trm2) state that the terminator processes $\mathbf{t}_n$ are in fact terminal arrows, with 0 as the terminal object. The laws (Mon1)-(Mon4) and (Sym1)-(Sym4) are the classical coherence conditions for a symmetric monoidal category, in the strict case where the the associativity isomorphisms are all identities. The laws (Com1)-(Com4) state that each object n has a cocommutative comonoid structure, with the terminator processes $\mathbf{t}_n : n \to 0$ as the counit and the duplicator processes $\mathbf{d}_n : n \to n+n$ as the comultiplication. Law (Com5) is an additional coherence condition that relates the comultiplications at different objects[1]

Buffering laws (Buf1)-(Buf2) state that for each object m, the input buffering operation is a left action of the monoid Buf(m) on the hom-set of processes from m to n. Laws (Buf3)-(Buf4) make the dual statement about output buffering. Law (Buf5) shows that input and output buffering are related to each other and to sequential composition in a kind of bimodule-like fashion. Laws (Buf6) and (Buf7) show how to distribute input and output buffering over parallel composition, and (Buf8) shows how the duplicator process $\mathbf{d}_n$ behaves with respect to buffering.

The feedback laws are not as easily characterized. Laws (Fbk1) and (Fbk2) state that feeding back multiple outputs to inputs can be achieved by feeding back one output at a time to an input. Law (Fbk3) shows that the generator process $\mathbf{g}_n\{\delta\}$ is already definable from the duplicator $\mathbf{d}_n$, input buffering, and feedback. (In spite of this, we include generators as basic standard processes because they play a useful role in the completeness proof.) Laws (Fbk4)-(Fbk6) show how to push sequential compositions in, out, and around feedback loops, and (Fbk7) is a similar type of property for parallel composition. Useless feedback loops can be eliminated with (Fbk8). Law (Fbk9), which is similar in spirit to (Buf8), shows how generator processes behave with respect to duplicators. Finally, (Fbk10) is a schema that gives some basic equalities between generators that cannot be proved from the the other laws. In it, β^i and β^j denote the i-fold and j-fold concatenation, respectively, of the buffer β with itself.

Our main result is the following:

Theorem 1 *The laws listed in Figures 5-8 are sound and complete for proving valid equations of dataflow calculus.*

It follows from the technique used in the proof that the validity problem for dataflow calculus is decidable.

The proof of Theorem 1 is extremely long—much too long to present here. For this reason, in the remainder of this paper we simply sketch the main ideas of the proof, pointing out along the way a number of interesting properties of dataflow calculus on which it relies.

[1] I wish to thank Barry Jay for helpful discussions concerning the independence of axiom (Com5), and the relationship of this issue to some comments made by Carboni and Walters [3] concerning results of T. Fox.

(Ext1)	$\mathbf{1}_n\{\delta\} = \mathbf{1}_n\delta$.
(Ext2)	$\mathbf{d}_n\{\delta, \delta'\} = \mathbf{d}_n(\delta \otimes \delta')$.
(Ext3)	$\mathbf{s}_{m,n}\{\eta, \delta\} = \mathbf{s}_{m,n}(\eta \otimes \delta)$.
(Ext4)	$f\{\delta\}g = f; (\delta g) = (f\delta); g$.
(Ext5)	$(f\circlearrowleft_p \{\delta\})\gamma = (f(\gamma \otimes \delta))\circlearrowleft_p$.

Figure 5: Extraction Laws

(Cat1)	$\mathbf{1}_m; f = f$.
(Cat2)	$f; \mathbf{1}_n = f$.
(Cat3)	$f; (g; h) = (f; g); h$.
(Trm1)	$\mathbf{t}_0 = \mathbf{1}_0$.
(Trm2)	$f; \mathbf{t}_n = \mathbf{t}_m$.
(Mon1)	$\mathbf{1}_{m+n} = \mathbf{1}_m \otimes \mathbf{1}_n$.
(Mon2)	$f \otimes \mathbf{1}_0 = f$.
(Mon3)	$(f \otimes g) \otimes h = f \otimes (g \otimes h)$.
(Mon4)	$(f \otimes f'); (g \otimes g') = (f; g) \otimes (f'; g')$.
(Sym1)	$\mathbf{s}_{n,0} = \mathbf{1}_n$.
(Sym2)	$\mathbf{s}_{m,n}; \mathbf{s}_{n,m} = \mathbf{1}_{m+n}$.
(Sym3)	$(f \otimes f'); \mathbf{s}_{n,n'} = \mathbf{s}_{m,m'}; (f' \otimes f)$, if $f : m \to n$ and $f' : m' \to n'$.
(Sym4)	$(\mathbf{1}_m \otimes \mathbf{s}_{n,p}); (\mathbf{s}_{m,p} \otimes \mathbf{1}_n) = \mathbf{s}_{m+n,p}$.
(Com1)	$\mathbf{d}_n; (\mathbf{1}_n \otimes \mathbf{t}_n) = \mathbf{1}_n$.
(Com2)	$\mathbf{d}_n; (\mathbf{t}_n \otimes \mathbf{1}_n) = \mathbf{1}_n$.
(Com3)	$\mathbf{d}_n; (\mathbf{1}_n \otimes \mathbf{d}_n) = \mathbf{d}_n; (\mathbf{d}_n \otimes \mathbf{1}_n)$.
(Com4)	$\mathbf{d}_n; \mathbf{s}_{n,n} = \mathbf{d}_n$.
(Com5)	$(\mathbf{d}_m \otimes \mathbf{d}_n); (\mathbf{1}_m \otimes \mathbf{s}_{m,n} \otimes \mathbf{1}_n) = \mathbf{d}_{m+n}$.
(Zer1)	$\mathbf{0}_{m+n} = \mathbf{0}_m \otimes \mathbf{0}_n$.
(Zer2)	$\mathbf{0}_{m+n}; \mathbf{s}_{m,n} = \mathbf{0}_{n+m}$.
(Zer3)	$\mathbf{0}_n; \mathbf{d}_n = \mathbf{0}_{n+n}$.

Figure 6: Categorical Laws

(Buf1)	$\langle\rangle f = f$.
(Buf2)	$\beta'(\beta f) = (\beta'\beta)f$.
(Buf3)	$f\langle\rangle = f$.
(Buf4)	$(f\gamma)\gamma' = f(\gamma\gamma')$.
(Buf5)	$(f\delta); g = f; (\delta g)$.
(Buf6)	$(\beta \otimes \beta')(f \otimes f') = (\beta f) \otimes (\beta' f')$.
(Buf7)	$(f\gamma) \otimes (f'\gamma') = (f \otimes f')(\gamma \otimes \gamma')$.
(Buf8)	$\beta\mathbf{d}_n = \mathbf{d}_n(\beta \otimes \beta)$.

Figure 7: Buffering Laws

(Fbk1)	$f\circlearrowleft_0 = f$.
(Fbk2)	$f\circlearrowleft_{p+q} = (f\circlearrowleft_p)\circlearrowleft_q$.
(Fbk3)	$(\delta\mathbf{d}_n)\circlearrowleft_n = \mathbf{g}_n\{\delta\}$.
(Fbk4)	$f; (g\circlearrowleft_p) = ((f \otimes \mathbf{1}_p); g)\circlearrowleft_p$.
(Fbk5)	$(f\circlearrowleft_p); g = (f; (g \otimes \mathbf{1}_p))\circlearrowleft_p$.
(Fbk6)	$(f; (\mathbf{1}_n \otimes g))\circlearrowleft_p = ((\mathbf{1}_m \otimes g); f)\circlearrowleft_q$, if $f : m + p \to n + q$ and $g : q \to p$.
(Fbk7)	$f \otimes (g\circlearrowleft_p) = (f \otimes g)\circlearrowleft_p$.
(Fbk8)	$((\mathbf{1}_{m+p} \otimes f); \mathbf{s}_{m,p+n})\circlearrowleft_m = (f \otimes \mathbf{1}_p); \mathbf{s}_{n,p}$, if $f : m \to n$.
(Fbk9)	$(\beta\mathbf{d}_n)\circlearrowleft_n; \mathbf{d}_n = (\beta\mathbf{d}_n)\circlearrowleft_n \otimes (\beta\mathbf{d}_n)\circlearrowleft_n$.
(Fbk10)	$(\beta^i\mathbf{d}_n)\circlearrowleft_n = (\beta^j\mathbf{d}_n)\circlearrowleft_n$, for all $i, j > 0$.

Figure 8: Feedback Laws

6 Sketch of the Proof

The soundness portion of the proof is accomplished primarily by exhibiting the required bisimulations. All the bisimulations are constructed in the obvious way, but the verification is tedious due to the number of transition rules. The proof is greatly facilitated by the fact that only the contents of buffers change as processes perform transitions—in particular processes do not change their syntactic structure.

The overall structure of the completeness proof is in two parts. The first part of the proof is to show that the equational laws listed in Figures 5-8 are sufficient to prove an arbitrary process equal to a "normal form." The normal form we use for a process $f : m \to n$ has the following structure:

$$(((\mathbf{1}_m \otimes v); (h; g)\circlearrowleft_p \{\delta\})\gamma,$$

where $v : p \to q$ is a parallel composition $v_0 \otimes v_1 \otimes \ldots \otimes v_s$ of the variables occurring in f, process $h : m+q \to r$ is a pure reticulate process, and $g : r \to n+p$ is constructed from basic standard processes $\mathbf{1}_1$ and $\mathbf{g}_1\{\eta\}$ using parallel composition only. In addition, a normal form must satisfy a technical condition, which says, informally, "every variable has some data path to an external output."

To show that an arbitrary process is provably equal to a normal form requires a substantial amount of work. A major portion of the work involves looking at the subclass of pure reticulate processes. Intuitively, we show that two pure reticulate processes are provably equal iff they "define the same input/output connections." More formally, we define the *connection function* of a reticulate $f : m \to n$ to be the partial function $\mathrm{Fun}_f : [n] \to [m]$ (note the contravariance) defined inductively as follows:

- $\mathrm{Fun}_{\mathbf{1}_n} = 1_n : [n] \to [n]$, the (total) identity function.
- $\mathrm{Fun}_{\mathbf{t}_n} = \emptyset : [0] \to [n]$, the empty partial function.
- $\mathrm{Fun}_{\mathbf{0}_n} = \emptyset : [n] \to [0]$, the empty partial function.
- $\mathrm{Fun}_{\mathbf{d}_n} : [n+n] \to [n]$ is defined by:

$$\mathrm{Fun}_{\mathbf{d}_n}(i) = \begin{cases} i, & \text{if } 0 \le i < n, \\ i-n, & \text{otherwise.} \end{cases}$$

- $\mathrm{Fun}_{\mathbf{s}_{m,n}} : [m+n] \to [n+m]$ is defined by:

$$\mathrm{Fun}_{\mathbf{s}_{m,n}}(i) = \begin{cases} i+n, & \text{if } 0 \le i < m, \\ i-m, & \text{otherwise.} \end{cases}$$

- $\mathrm{Fun}_{\beta f} = \mathrm{Fun}_f$ and $\mathrm{Fun}_{f\gamma} = \mathrm{Fun}_f$.
- $\mathrm{Fun}_{f;g} = \mathrm{Fun}_f \circ \mathrm{Fun}_g$.
- If $f : m \to n$ and $f : m' \to n'$, then $\mathrm{Fun}_{f \otimes f'} : [n+n'] \to [m+m']$ is defined by:

$$\mathrm{Fun}_{f \otimes f'}(i) = \begin{cases} \mathrm{Fun}_f(i), & \text{if } 0 \le i < m, \\ \mathrm{Fun}_{f'}(i-m)+n, & \text{otherwise.} \end{cases}$$

Intuitively, $\mathrm{Fun}_f(j) = i$ iff the process f transmits to its jth output all data arriving on its ith input. This intuition is validated for our operational semantics by defining the *input/output relation* of f to be the relation $\mathrm{Rel}_f \subseteq \mathrm{Buf}(m) \times \mathrm{Buf}(n)$ consisting of all pairs (β, γ) such that there exists an internal computation of the form:

$$\beta((\langle\rangle f)\langle\rangle) \xRightarrow{\#^*} \langle\rangle(f'\gamma),$$

and then establishing that Fun_f and Rel_f correspond in a natural way. We also show that buffer bisimilar processes have identical input/output relations. We prove the following completeness result for the class of pure reticulate processes:

Lemma 1 *For pure reticulate processes $f, f' : m \to n$, we have $\mathrm{Fun}_f = \mathrm{Fun}_{f'}$ iff the equation $f = f'$ is provable from the categorical laws.*

This is accomplished by considering subclasses of "permutative," "duplicative," "contractive," and "expansive" processes (containing, respectively, processes **s**, **d**, **t**, and **0**, together with identity processes **1**), proving normal form and completeness results for each class, and then combining these results using lemmas that show how the various classes commute with each other.

Next, the completeness result for pure reticulate processes is extended to the full class of reticulate processes. This is done by defining a notion of the *buffer contents* $\mathrm{Buf}_f \in \mathrm{Buf}(n)$ of a process f in an inductive fashion analogous to the definition of Fun_f, and then proving:

Lemma 2 *For reticulate processes $f, f' : m \to n$, we have $\mathrm{Fun}_f = \mathrm{Fun}_{f'}$ and $\mathrm{Buf}_f = \mathrm{Buf}_{f'}$ iff the equation $f = f'$ is provable from the categorical laws and the buffering laws.*

This result is then extended once again, to the class of all feedback-free standard processes. For this, we extend the notion Buf_f to apply to generative processes, whose "buffer contents" can be infinite. For this result, we need to make use of the extraction laws and the feedback laws (Fbk3), (Fbk9), and (Fbk10).

To extend the completeness result for feedback-free standard processes to the class of all standard processes, we use a technique of "feedback elimination."

Actually, we don't eliminate feedback completely, we merely show how feedback loops in standard processes can be made very small and ultimately hidden inside generative processes $\mathbf{g}_1\{\eta\}$. This technique allows us to show that any standard process can be proved equal to one of the form $(h; g)\gamma$, where h is a pure reticulate process, and g is constructed from $\mathbf{1}_1$ and $\mathbf{g}_1\{\eta\}$ using parallel composition only. The feedback elimination technique requires the following expressiveness result for pure reticulate processes:

Lemma 3 *Suppose $\phi : [n] \to [m]$ is a partial function. Then there exists a pure reticulate process $f : m \to n$ such that $\mathrm{Fun}_f = \phi$.*

We then have:

Lemma 4 *For standard processes $f, f' : m \to n$, we have $\mathrm{Fun}_f = \mathrm{Fun}_{f'}$ and $\mathrm{Buf}_f = \mathrm{Buf}_{f'}$ iff the equation $f = f'$ is provable from the full set of laws in Figures 5-8.*

Finally, we admit nonstandard processes, and obtain the general normal form result mentioned above.

The second main part of the completeness proof consists of showing: (1) that if f and f' are normal forms that are *essentially identical* (*i.e.* "identical up to permutation of their variables"), then they are provably equal, and (2) that if f and f' are not essentially identical, then there is an interpretation of their variables under which they have distinct input/output relations, hence under which they are not buffer bisimilar. It follows from these two assertions that if normal forms f and f' are buffer bisimilar under all interpretations, then they are provably equal. The proof of (1) is reasonably straightforward, given our completeness results for the class of standard processes and the expressiveness result for the class of pure reticulate processes.

The proof of (2) is more difficult. For it, we use an interpretation of a standard form, in which each variable is interpreted as a process that first makes a certain indeterminate choice, and then subsequently performs a "merging and tagging" function in which arriving data values are tagged to indicate on which input port they arrived, then merged into a single sequence, and then output on all output ports after tagging once again to indicate the output port on which they were emitted. The interpretation depends both on the set X of variables appearing in f and f' and also on the set D of data values appearing in them.

More precisely, suppose normal forms $f : m \to n$ and $f' : m \to n$ (in the generic dataflow calculus $\mathcal{D}(X)$ determined by a finite set of process variables X) are given. Let D be the set of all data values appearing in buffers either in f or in f'. Let $X_{\mathcal{N}}$ denote the set $X \times \mathcal{N}$. We will use the notation x_a to denote an element (x, a) of $X_{\mathcal{N}}$. We construct a dataflow calculus $\mathcal{S}(X, D)$ having as its set of basic nonstandard processes the disjoint union $X + X'$, where

$$X' = \{x_a\{\delta\} : x \in X, a \in \mathcal{N}, \delta \in \mathrm{Buf}(\mathrm{owd}(x))\},$$

and $x_a\{\delta\} \in X'$ has the same input width and the same output width as $x \in X$. Here each expression $x_a\{\delta\}$ is to be regarded as a single symbol denoting a basic nonstandard process, in the same way, for example, that each expression $\mathbf{1}_n\{\delta\}$ is regarded as single symbol denoting a basic standard process. We choose nonstandard transition axioms that yield the following behavior for a variable $x : m_x \to n_x$ in X:

1. Choose, in an indeterminate fashion, a value a as an "ID." Then become the process $x_a\{\delta_a\}$, where δ_a is a buffer of width n_x that contains an encoded version of the value x_a as its jth component, for all $j \in n_x$. These values, which will be the first to be output by $x_a\{\delta_a\}$, serve to announce the indeterminate choice that was made, and also to exercise all data paths leading from $x_{\{}\delta_a\}$.

2. The process $x_a\{\delta_a\}$ then begins a tagging and merging function, in which values arriving on input port i are tagged by i and placed in the internal buffer for eventual output on all output ports. Interleaved with these transitions are the actual output transitions, in which a value at the head of the internal buffer for output j is removed from the buffer, tagged by j, and output on port j.

The idea behind this interpretation is that data values flowing around the feedback loop will accumulate an encoded history of the paths they take. When these values reach an external output, some of the internal network structure will become observable. The main work in this part of the completeness proof is to show that, under suitable conditions, in fact all the relevant internal structure of a normal form can be made observable at the external outputs.

We comment briefly on the role of the ID's in this interpretation, which might otherwise seem somewhat mysterious. A significant obstacle in trying to construct an interpretation that distinguishes normal forms f and f' that are not essentially identical is the following: f and f' may have more than one occurrence of each variable, but an interpretation has to assign the same process to all occurrences of a variable. This potentially limits our ability to discover the internal structure of f and f', because it might be hard to tell apart values originating at two different occurrences of the same variable x. To get around this problem, we interpret a variable x as a process that initially makes an indeterminate selection of an ID a, to become a process x_a. The choice is announced on all outputs as it is made. If all occurrences of variables choose different ID's, then distinguishing which values come from which variables is very easy. Furthermore, if we have the set of outputs produced in all possible computations of f and f', then it is possible to distinguish those in which all variables chose different ID's from those in which some variables chose the same ID's. We may then concern ourselves only with the former type of computations.

To give the nonstandard transition axioms for $\mathcal{S}(X, D)$ we must be more specific about how tagging is performed. Since the set D is finite, but the set $\mathcal{N}$ is countably infinite, using standard coding techniques we can assign:

1. a value $* \in \mathcal{N}$,
2. to each variable $x \in X$ a value $\ulcorner x \urcorner \in \mathcal{N}$,
3. to each $x_a \in X_{\mathcal{N}}$ a value $\ulcorner x_a \urcorner \in \mathcal{N}$,

in such a way that the sets D, $\{*\}$, $\{\ulcorner x \urcorner : x \in X\}$, and $\{\ulcorner x_a \urcorner : x_a \in X_{\mathcal{N}}\}$ are pairwise disjoint. Moreover, we can choose a "pairing function" that takes each pair (i, v), where $i \in \mathrm{Ord}$ and $v \in \mathcal{N}$ to a *tagged value* $(i :: v) \in \mathcal{N}$, together with partial functions

$$\begin{aligned} \mathrm{val}: &\quad \mathcal{N} \to D + \{*\} + X + X_{\mathcal{N}} \\ \mathrm{tags}: &\quad \mathcal{N} \to \mathrm{Ord}^*, \end{aligned}$$

in such a way that the set $\{(i :: v) : i \in \mathrm{Ord} \text{ and } v \in \mathcal{N}\}$ is disjoint from each of D, $\{*\}$, $\{\ulcorner x \urcorner : x \in X\}$, and $\{\ulcorner x_a \urcorner : x_a \in X_{\mathcal{N}}\}$, such that

1. $\mathrm{val}(v) = v$ for all $v \in D + \{*\}$, $\mathrm{val}(\ulcorner x \urcorner) = x$ for all $x \in X$, and $\mathrm{val}(\ulcorner x_a \urcorner) = x_a$ for all $x_a \in X_{\mathcal{N}}$,
2. $\mathrm{tags}(v) = \epsilon$ for all $v \in D + \{*\}$, $\mathrm{tags}(\ulcorner x \urcorner) = \epsilon$ for all $x \in X$, and $\mathrm{tags}(\ulcorner x_a \urcorner) = \epsilon$ for all $x_a \in X_{\mathcal{N}}$,
3. $\mathrm{tags}(i :: v) = i.\mathrm{tags}(v)$ for all $i \in \mathrm{Ord}$ and $v \in \mathcal{N}$,

and such that val and tags are undefined in all other cases.

Then $\mathcal{S}(X, D)$ has the following as its set of nonstandard transition axioms:

$$x \overset{0\#a}{\Longrightarrow} x_a\{\langle 0 : \ulcorner x_a \urcorner, 1 : \ulcorner x_a \urcorner, \ldots, (n_x - 1) : \ulcorner x_a \urcorner \rangle\}$$

$$x_a\{\delta\} \overset{i?v}{\Longrightarrow} x_a\{\langle 0 : (i :: v), \ldots, (n_x - 1) : (i :: v)\rangle \delta\}$$

$$x_a\{\delta\langle j : v\rangle\} \overset{j!(j::v)}{\Longrightarrow} x_a\{\delta\}.$$

The standard interpretation $I : \mathcal{D}(X) \to \mathcal{S}(X, D)$ maps each variable $x \in X$ to the corresponding basic nonstandard process x of $\mathcal{S}(X, D)$.

To show that two processes $f : m \to n$ and $f' : m \to n$ have distinct input/output relations under a standard interpretation, it is sufficient to show that for some particular input buffer $\iota \in \mathrm{Buf}(m)$, the processes f and f' produce distinct sets of output when supplied with input ι. It turns out that it is sufficient to define ι as follows:

$$\iota = \langle 0 : (0 :: *), 1 : (1 :: *), \ldots, (m-1) : ((m-1) :: *)\rangle.$$

Thus, at each input $i \in m$, we place the special marker value $*$, after first "tagging" it with $i \in \mathrm{Ord}$ so that we can later determine the origin of any particular instance of $*$.

To make the final connection between syntax and semantics and complete the proof, we establish two technical properties about computations in dataflow calculus, which we mention here only informally. The first property is an invariant, which states that only "correct" information about the syntactic structure of a normal form f accumulates in the tags of data values as computation progresses. The second property is a progress property, which states that if f and f' are normal forms that are not essentially identical, then one of them has a finite computation, in which enough information about its internal structure is output, that it is impossible for the other to do likewise.

In the proof of these properties, we make use of two helpful results about computations in dataflow calculus. The first is a kind of "conditional confluence" result. Let us call transitions $\Gamma : f \Longrightarrow g$ and $\Gamma' : f \Longrightarrow g'$ *independent* if either they have different modes, or else they have the same modes but different ports. Then we have the following:

Lemma 5 *Suppose transitions $\Gamma : f \Longrightarrow g$ and $\Gamma' : f \Longrightarrow g'$ are independent, and that one of them is a standard transition. Then there exists a unique process h, and unique transitions $\Delta : g' \Longrightarrow h$ and $\Delta' : g \Longrightarrow h$, such that Δ has the same mode and token as Γ, and Δ' has the same mode and token as Γ'.*

From this result, one can show that every process has a "buffer normalizing computation," leading to a unique *buffer normal form*, in which all data values other than those inside of generative processes and basic nonstandard processes, have been moved as "far to the right" as possible. Buffer normalizing computations contain only standard transitions that are inferred without using the axioms for generative processes, and without using the feedback rule in which output in the feedback buffer is recycled as input. The above result also enables us to define a kind of "permutation equivalence" of computations (*c.f.* [13]). Using this notion, we can state our second useful result: a kind of "standardization theorem," which shows that any computation is permutation equivalent to one in a standard form. This standardization theorem greatly simplifies the proof of our invariance result.

Lemma 6 *Suppose*

$$f_0 \Longrightarrow f_1 \Longrightarrow f_2 \Longrightarrow \dots \Longrightarrow f_k$$

is a computation for a process f_0 in a dataflow calculus. Then there exists a permutation equivalent computation

$$f_0 \overset{\#^*}{\Longrightarrow} f'_0 \Longrightarrow g_1 \overset{\#^*}{\Longrightarrow} f'_1 \Longrightarrow g_2 \overset{\#^*}{\Longrightarrow} f'_2 \Longrightarrow \dots \overset{\#^*}{\Longrightarrow} f'_k,$$

such that, for each i, the computation $f_i \overset{\#^*}{\Longrightarrow} f'_i$ is a buffer normalizing computation, and f'_i is the buffer normal form of f_i.

The proof of Theorem 1 we have sketched above uses the fact that the set of data values is infinite, so that standard coding techniques can be used for the tagging of data values. This is not essential, since the argument can be modified to apply to dataflow calculi with only finite sets of data values, as long as these sets can be arbitrarily large. However, the proof does depend in an essential way on the ability of the processes to make indeterminate choices. Note that there are two ways in which indeterminate choice is used by the basic nonstandard processes in the above interpretation: first, in choosing their ID's, and second, in performing the merging function. It is not clear whether the techniques can be extended to give a completeness theorem in the case where the interpretations are restricted to "determinate" dataflow calculi.

References

[1] J. C. M. Baeten and W. P. Weijland. *Process Algebra*, volume 18 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, 1990.

[2] J. D. Brock and W. B. Ackerman. Scenarios: A model of non-determinate computation. In *Formalization of Programming Concepts*, volume 107 of *Lecture Notes in Computer Science*, pages 252–259. Springer-Verlag, 1981.

[3] A. Carboni and R. F. C. Walters. Cartesian bicategories I. *Journal of Pure and Applied Algebra*, 49:11–32, 1987.

[4] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information Processing 74*, pages 471–475. North-Holland, 1974.

[5] G. Kahn and D. B. MacQueen. Coroutines and networks of parallel processes. In B. Gilchrist, editor, *Information Processing 77*, pages 993–998. North-Holland, 1977.

[6] J. Lambek and P. J. Scott. *Introduction to Higher-Order Categorical Logic*, volume 7 of *Cambridge Studies in Advanced Mathematics.* Cambridge University Press, 1986.

[7] N. A. Lynch and E. W. Stark. A proof of the Kahn principle for input/output automata. *Information and Computation*, 82(1):81–92, July 1989.

[8] R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[9] D. M. R. Park. Concurrency and automata on infinite sequences. In *Theoretical Computer Science*, volume 104 of *Lecture Notes in Computer Science.* Springer-Verlag, 1981.

[10] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, 1981.

[11] E. W. Stark. Concurrent transition system semantics of process networks. In *Fourteenth ACM Symposium on Principles of Programming Languages*, pages 199–210, January 1987.

[12] E. W. Stark. Compositional relational semantics for indeterminate dataflow networks. In *Category Theory and Computer Science*, volume 389 of *Lecture Notes in Computer Science*, pages 52–74. Springer-Verlag, Manchester, U. K., 1989.

[13] E. W. Stark. Connections between a concrete and an abstract model of concurrent systems. In *Fifth Conference on the Mathematical Foundations of Programming Semantics*, volume 442 of *Lecture Notes in Computer Science*, pages 53–79. Springer-Verlag, New Orleans, LA, 1990.

[14] E. W. Stark. A simple generalization of Kahn's principle to indeterminate dataflow networks. In M. Z. Kwiatkowska, M. W. Shields, and R. M. Thomas, editors, *Semantics for Concurrency, Leicester 1990*, pages 157–176. Springer-Verlag, 1990.