The algebra of stream processing functions

Manfred Broy\textsuperscript{a,*}, Gheorghe Ștefănescu\textsuperscript{b}

\textsuperscript{a} Institute of Informatics, Technische Universität München, D-80290 München, Germany
\textsuperscript{b} Faculty of Mathematics, University of Bucharest, Str. Academiei 14, RO-70109 Bucharest, Romania

Received July 1996; revised May 1999
Communicated by M. Wirsing

Abstract

Data flow networks are a model of concurrent computation. They consist of a collection of concurrent asynchronous processes which communicate by sending data over FIFO channels. In this paper we study the algebraic structure of the data flow networks and base their semantics on stream processing functions. Our algebraic theory is based on the calculus of flownomials. With an additive (or cantorian) interpretation the calculus gives a unified presentation of the classical algebraic models for control structures, that is, regular algebra and iteration theories. The kernel of the calculus is an equational axiomatization called basic network algebra (BNA) for flow graphs modulo graph isomorphism. We show that the algebra of stream processing functions called SPF (used for deterministic networks) and the algebra of sets of stream processing functions called SPF (used for nondeterministic networks) are BNA algebras. Actually they give a multiplicative (or cartesian) interpretation of the calculus of flownomials. As a byproduct this shows that both semantic models are compositional. This means the semantics of a network may be described in terms of the semantics of its components. (As it is well known this is not true for the input–output relational semantics of nondeterministic networks.) We also identify additional axioms satisfied by the branching constants in these two algebraic theories. For the deterministic case we study in addition the coarser equivalence relation on networks given by the input–output behavior and provide a correct and complete axiomatization. © 2001 Elsevier Science B.V. All rights reserved.

1. Introduction

The idea of control and data flow is a classic concept that can be found in most approaches to computation, programming, and computing machinery. Often the flow
is visualized by flow graphs. The idea of data flow had mainly two sources. Single assignment languages (see [36] are based on the concept of a set of (nonrecursive) declarations. The order of the evaluation of the declarations is then only determined by their data dependencies. These dependencies can be shown in an acyclic graph called their data flow graph. Influenced by these ideas and by the concept of Petri-nets and their firing rules, Jack Dennis [16] suggested data flow graphs and gave firing rule semantics for them. Quite independently, versions of data flow graphs can be found in many software engineering methods and also for the description of switching circuits.

Gilles Kahn suggested a mathematical model for asynchronously communicating agents that can be used as a model for deterministic data flow nets. The data flow networks used in [22] describe a collection of processes which work in a parallel and asynchronous way and communicate by sending values over FIFO channels. Moreover, Kahn’s data flow networks were deterministic and thus the input–output relation specified by such processes is actually a (continuous) function. The main result of Kahn in [22] asserts that the function specified by a deterministic network may be obtained from the functions specified by its components using a least fixed-point construction.

It turns out that Kahn’s elegant theorem cannot be extended in an easy way to the case of nondeterministic data flow networks. In such networks, the components are capable of making arbitrary choices during computation and the input–output behavior specified by such a network is not longer a function, but an arbitrary relation. For such networks, fundamental results by Keller [23] and Brock-Ackermann [7] have shown a mismatch between the operational meaning of the networks and their input–output behavior. In other words, the input–output behavior of the components of a network is no longer sufficient to compute its whole behavior. This situation, known as merge or Brock–Ackermann anomaly, was solved by adding information to the input–output behavior, e.g. using scenarios, traces, oracles, etc. Extensions of the theory of data flow networks to the nondeterministic case were suggested for instance in [31,9,26,20,11].

In this paper we take the viewpoint of [28,9] and model nondeterministic data flow networks with the help of oracles. An oracle provides a priori global information on the choices in all the nondeterministic points of a network and it allows giving the semantics of a nondeterministic network by a set of (stream processing) functions.

After this short and informal presentation of data flow networks we now discuss their algebraic counterpart. Graphs are used in many methods in computing science to represent the flow of information, data, and control. To be able to apply algebraic techniques when dealing with such graphs, we have to represent graphs by terms. To do this, we have to find appropriate algebraic operators for the construction of graphs. Typically, the same graphs (isomorphic graphs) can then be represented by quite different terms. Two terms that denote the same graph are, therefore, called graph isomorphic. The graph isomorphism on terms is an equivalence relation on terms that can be axiomatized by equations. In addition to these laws of graph isomorphism, we use more specific laws that hold due to the semantic theories of the specific flow models.
Algebraic models for nondeterministic data flow cannot naively be obtained as extensions of those for deterministic data flow, mainly due to the unsoundness of the classical unfold techniques for the fixed-point equation. This fixed-point equation provides the fundamental technique for defining the semantics of cyclic deterministic networks. It corresponds to defining the semantics of a cyclic process by unfolding the loop. Let $a$ and $b$ be types and $a + b$ be the type of pairs of elements of types $a$ and $b$. Formally, for a function $f : a + b \rightarrow b$ the fixed-point equation that describes the feedback of the output of $f$ to its second input may be written as

$$(f \cdot \mathcal{R}^b)^\uparrow b = \mathcal{R}^a \cdot (I_a \star (f \cdot \mathcal{R}^b)^\uparrow b) \cdot f$$

(see Section 3 for more details, including the meaning of the operators used in the above formula). This function equation corresponds to the semantic equivalence of the two graphs given in Fig. 1. What is important here is the observation that the left-hand side of the equation contains an occurrence of $f$, while the right-hand side has two such occurrences. The corresponding networks are not isomorphic. Indeed, if $f$ is nondeterministic, then different behaviors may be selected for the two occurrences of $f$ in the right-hand side term, while a unique choice for $f$ may be done in the left-hand side term and the resulting terms may be different.\footnote{By a similar argument, the strong commutation axiom S4 in Table 1 is valid for deterministic networks, but not for nondeterministic ones.} To cope with this problem we use the calculus of flownomials, see [35]. This calculus uses a technique that is different to the one needed for nondeterministic data flow to model the cyclic processes.

The calculus of flownomials is an algebraic calculus very similar to the calculus of polynomials. Its aim is to capture the syntax and the semantics of several graph-like models used in computer science. It was obtained as a unification of the classical...
regular algebras presented in [25,15] and of the iteration theories developed starting
with the study of flowchart schemes in [17,5,32,34,14] among other. See [35] for the
basic results of the calculus and some historical comments.

In order to obtain an axiomatization for cyclic processes one has to use a looping
operator. Such a looping operator allows connecting an output with an input. The
main novelty of the calculus of flownomials consists in using a new axiomatic looping
operation, called the feedback operator in [34]. The key feature of this operation is that
(1) after its application both the connected inputs and outputs are hidden (they are not
visible anymore).

Some other possibilities are to use the repetition in [25,33], where
(2) after the application of this operation both the connected inputs and outputs remain
visible,

or iteration in [17], where
(3) after the application of this operation the connected inputs remain visible, but not
the outputs.\(^2\)

An analysis shows that only the feedback operator allows for an (easy and natu-
ral) axiomatization of the cyclic processes in the graph-isomorphism setting. The other
looping operators such as repetition and iteration require a sort of fixed-point axioma-
tization which, as we mentioned above, departs from the graph-isomorphism setting.

The kernel of the flownomial calculus is given by the ax-flow algebra; we also
use the BNA (basic network algebra) acronym of [2] for the corresponding equational
theory. This algebra gives a complete axiomatization for flow graphs/networks modulo
graph isomorphism. For a detailed treatment see [34,13,35].

One aim of the present paper is to show that the flownomial calculus may be ap-
plied to the study of (asynchronous) data flow computation as well. To this end we
use a different “multiplicative” (or “cartesian”) interpretation of flownomials. As we
said, what we study here from the various approaches to handle the semantics of non-
deterministic data flow networks are the algebraic properties of the oracle-based model
presented in [28,9,11]. In this approach, the semantics of a nondeterministic data flow
network is specified as a set of stream processing functions.

The main new results of our paper are as follows:

• We show that the algebra of stream processing functions called SPF (which we
use as a semantic model for deterministic networks) and the algebra of sets of
stream processing functions called PSPF (which we use as a semantic model for
nondeterministic networks) are both BNA models. As a byproduct, these results
show that both semantics above are compositional. This means the semantics of a
network may be described in terms of the semantics of its components. (As it is well
known this is not true for the input–output relational semantics of nondeterministic

\(^2\)In [11] a “feedback” operation different from the one in this paper is used. In fact, the operation in [11]
is a dual iteration, where after the application of the operation the connected outputs remain available, but
not the inputs.
networks.) We also identify additional axioms satisfied by the branching constants in these two algebras.

• For the deterministic case we also study the coarser equivalence on networks given by the input–output behavior and provide a correct and complete axiomatization. This theorem is based on the axiomatization of flow graphs with respect to the unfolding equivalence published in [33] and Chapter 8 of [35]. What is new here is the theorem regarding the coincidence of the input–output behavior and the unfolding equivalence in the case of deterministic data flow networks.

A somewhat similar approach is given by Stark. In [30] it is shown that an algebra with the same operators (parallel and sequential compositions and feedback) may be used to study nondeterministic data flow networks. As branching constants Stark uses the ‘copy’ constant and certain sink and source constants. The main result of [30] is a theorem of correctness and completeness for networks modulo “buffer bisimilarity”.

The paper is organized as follows: In Section 2 we give a short overview of the calculus of flownomials. Section 3 is devoted to the study of deterministic data flow networks. We give two complete axiomatizations presented as extensions of BNA axioms, namely one for networks modulo graph isomorphism equivalence and one for the coarser equivalence induced on networks by the input–output behavior. Section 4 deals with nondeterministic data flow networks. We show that the \(\mathcal{P}_{SPF}\) (sets of stream processing functions) model satisfies the BNA axioms as well. Some additional sound laws are given. However, the problem of a complete axiomatization for the equivalence induced on networks by the \(\mathcal{P}_{SPF}\) semantics is not solved and left open. Detailed proofs of certain technical theorems are presented in Section 5.

2. Flownomials

The algebra of flownomials gives an algebraic presentation of directed flow graphs and their behaviors. In the standard version of [35] it uses three operations:

“\(\bigodot\)” (parallel composition), “\(\cdot\)” (sequential composition) and “\(\uparrow\)” (feedback) and various constants for describing the branching structure of the flow graphs such as “\(I_a\)” (identity), “\(X^b\)” (transposition), “\(\wedge^a\)” (ramification) and “\(\vee^k\)” (identification).

Note that \(k\) is a natural number specifying the branching degree, while \(a\) and \(b\) model the type of the network input or output interfaces.

Table 1 lists the groups of axioms for cantorian flownomials we are starting with. In Table 1 we use some particular cases of the ramification and identification constants, namely \(\wedge_a^0, \wedge_a^2, \vee_a^0, \vee_a^2\) denoted by \(\bot^a, \wedge^a, \top_a, \vee_a\), respectively. The general branching constants may be easily defined in terms of these particular ones; see, e.g. [35]. The adapted axioms for cartesian flownomials used to model data flow networks will be given in Table 2.

In the standard version presented in [35] there are three groups of algebraic equations (see Table 1):
The standard axioms for the calculus of (cantonian) polynomials

Table 1
The standard axioms for the calculus of (cantonian) polynomials

I. Axioms for ssms-ies (symmetric strict monoidal categories)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>B1</td>
<td>$f \star (g \star b) = (f \star g) \star b$</td>
</tr>
<tr>
<td>B2</td>
<td>$I_0 \star f = f = f \star I_0$</td>
</tr>
<tr>
<td>B3</td>
<td>$f \cdot (g \cdot h) = (f \cdot g) \cdot h$</td>
</tr>
<tr>
<td>B4</td>
<td>$I_0 \cdot f = f = f \cdot I_0$</td>
</tr>
<tr>
<td>B5</td>
<td>$(f \star f') \cdot (g \star g') = f \cdot g \star f' \cdot g'$</td>
</tr>
<tr>
<td>B6</td>
<td>$I_0 \star I_0 = I_{a+b}$</td>
</tr>
<tr>
<td>B7</td>
<td>$\alpha x^b \cdot \alpha x^a = I_{a+b}$</td>
</tr>
<tr>
<td>B8</td>
<td>$\alpha x^b = (\alpha x^b \star I_0) \cdot (I_0 \star \alpha x^a)$</td>
</tr>
<tr>
<td>B9</td>
<td>$\alpha x^{b+c} = (\alpha x^b \star I_0) \cdot (I_0 \star \alpha x^c)$</td>
</tr>
</tbody>
</table>

II. Axioms for feedback

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>$f \cdot (g \uparrow) \cdot h = ((f \star I_0) \cdot g \cdot (h \star I_0)) \uparrow^c$ (relating “$\uparrow$” and “$\uparrow^c$”)</td>
</tr>
<tr>
<td>R2</td>
<td>$f \star g \uparrow^c = (f \star g) \uparrow^c$ (relating “$\uparrow$” and “$\uparrow^c$”)</td>
</tr>
<tr>
<td>R3</td>
<td>$(f \cdot (I_0 \star g)) \uparrow^c = ((I_0 \star g) \cdot f) \uparrow^c$ (shifting blocks on feedback)</td>
</tr>
<tr>
<td>R4</td>
<td>$f \uparrow^0 = f$ (no feedback)</td>
</tr>
<tr>
<td>R5</td>
<td>$(f \uparrow^b) \uparrow^a = f \uparrow^a+b$ (multiple feedbacks)</td>
</tr>
<tr>
<td>R6</td>
<td>$I_0 \uparrow^a = I_0$ (feedback on identity)</td>
</tr>
<tr>
<td>R7</td>
<td>$\alpha x^a \uparrow^a = I_0$ (feedback on transposition)</td>
</tr>
</tbody>
</table>

III. Axioms for the additional constants $\top, \bot, \lor, \land$ (without feedback)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>$(\land_a \star I_0) \cdot \land_a = (I_0 \star \land_a) \cdot \land_a$</td>
</tr>
<tr>
<td>A2</td>
<td>$\alpha x^a \cdot \land_a = \land_a$</td>
</tr>
<tr>
<td>A3</td>
<td>$(\land_a \star I_0) \cdot \land_a = I_0$</td>
</tr>
<tr>
<td>A4</td>
<td>$\lor_a \cdot \bot_a = \bot_a \star \land_a$</td>
</tr>
<tr>
<td>A5</td>
<td>$\land_a \cdot (\land_a \star I_0) = \land_a \cdot (I_0 \star \land_a)$</td>
</tr>
<tr>
<td>A6</td>
<td>$\land_a \cdot \land_a = \land_a$</td>
</tr>
<tr>
<td>A7</td>
<td>$\land_a \cdot (\land_a \star I_0) = I_0$</td>
</tr>
<tr>
<td>A8</td>
<td>$\land_a \cdot \bot_a = \bot_a \star \land_a$</td>
</tr>
<tr>
<td>A9</td>
<td>$\land_a \cdot \land_a = I_0$</td>
</tr>
<tr>
<td>A10</td>
<td>$\land_a \cdot (I_0 \star \land_a) \cdot (I_0 \star \alpha x^a \star I_0) \cdot (\land_a \star \land_a) \cdot (\land_a \star \land_a) = \land_a \cdot (\land_a \star \land_a) \cdot (I_0 \star \land_a) \cdot (I_0 \star \land_a)$</td>
</tr>
<tr>
<td>A11</td>
<td>$\land_a \cdot \land_a = I_0$</td>
</tr>
<tr>
<td>A12</td>
<td>$\top_0 = I_0$</td>
</tr>
<tr>
<td>A13</td>
<td>$\top_0 = I_0$</td>
</tr>
<tr>
<td>A14</td>
<td>$\lor_0 = I_0$</td>
</tr>
<tr>
<td>A15</td>
<td>$\land_0 \cdot (I_0 \star \land_a) \cdot (\land_a \star \land_a) \cdot (\land_a \star \land_a) = \land_0 \cdot (\land_a \star \land_a) \cdot (I_0 \star \land_a) \cdot (I_0 \star \land_a)$</td>
</tr>
<tr>
<td>A16</td>
<td>$\land_0 = I_0$</td>
</tr>
<tr>
<td>A17</td>
<td>$\land_0 = I_0$</td>
</tr>
<tr>
<td>A18</td>
<td>$\land_0 = I_0$</td>
</tr>
<tr>
<td>A19</td>
<td>$\land_0 = I_0$</td>
</tr>
</tbody>
</table>

IV. Axioms for the action of feedback on the additional constants

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>F3</td>
<td>$\land_a \uparrow^a = \land_a$</td>
</tr>
<tr>
<td>F4</td>
<td>$\land_a \uparrow^a = \top_a$</td>
</tr>
<tr>
<td>F5</td>
<td>$(I_0 \star \land_a) \cdot (\alpha x^a \star I_0) \cdot (I_0 \star \land_a) \cdot (\land_a \star \land_a) \cdot (\land_a \star \land_a) = I_0$</td>
</tr>
</tbody>
</table>

V. The strong axioms $(f : a \rightarrow b)$

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>$\top_a \cdot f = \top_b$</td>
</tr>
<tr>
<td>S2</td>
<td>$\land_a \cdot f = (f \star f) \cdot \land_b$</td>
</tr>
<tr>
<td>S3</td>
<td>$f \cdot \land_a = \land_a$</td>
</tr>
<tr>
<td>S4</td>
<td>$f \cdot \land_a = (f \star f) \cdot \land_a$</td>
</tr>
</tbody>
</table>

VI. The enzymatic rule

Let $E$ be a class of abstract relations (i.e., of terms written with $\star, \cdot, I, X$ and some constants in $\top, \bot, \lor, \land$)

ENZ$_E$: if $f : a + c \rightarrow b + c$ and $g : a + d \rightarrow b + d$ there exists $r : c \rightarrow d$ in $E$ such that $f \cdot (I_0 \star r) = (I_0 \star r) \cdot g$, then $f \uparrow^c = g \uparrow^d$

(A) a large group of algebraic equations for flow graphs modulo graph isomorphism B1–B10, A1–A19, R1–R5, and F1–F5;
(S) some critical algebraic equations S1–S4 for ramification and identification data flow nodes;
(Z) an axiom scheme ENZ, presented as a conditional equation.
Table 2

Additional axioms for deterministic data flow networks

III. Axioms for the additional constants $\nabla^a$, $\check{\flat}$ (without feedback)

<table>
<thead>
<tr>
<th>Axiom</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A5</td>
<td>$\check{\flat}^a \cdot (\check{\flat}^a \star l_0) = \check{\flat}^a \cdot (l_0 \star \check{\flat}^a)$</td>
</tr>
<tr>
<td>A7</td>
<td>$\check{\flat}^a \cdot (\nabla^a \star l_0) = l_0$</td>
</tr>
<tr>
<td>A9</td>
<td>$\nabla^a \cdot \nabla^a = l_0$</td>
</tr>
<tr>
<td>A12</td>
<td>$\nabla^a = l_0$</td>
</tr>
<tr>
<td>A13</td>
<td>$\nabla^a \star l_0 = l_0 \star \nabla^a$</td>
</tr>
<tr>
<td>A16</td>
<td>$\nabla^0 = l_0$</td>
</tr>
<tr>
<td>A17</td>
<td>$\nabla^a \star \nabla^b = (\check{\flat}^a \star \check{\flat}^b \star (l_0 \star X^b \star l_0))$</td>
</tr>
<tr>
<td>A18</td>
<td>$\check{\flat}^0 = l_0$</td>
</tr>
<tr>
<td>A19</td>
<td>$\check{\flat}^0 \star \check{\flat}^0 = \check{\flat}^0 \star \check{\flat}^0$</td>
</tr>
</tbody>
</table>

IV. Axioms for the action of feedback on the branching constants

<table>
<thead>
<tr>
<th>Axiom</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F4</td>
<td>$\check{\flat}^a \uparrow \check{\flat}^a = \nabla^a$</td>
</tr>
</tbody>
</table>

V. The strong axioms $(f : a \rightarrow b)$

<table>
<thead>
<tr>
<th>Axiom</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>S3</td>
<td>$f \cdot \check{\flat}^b = \check{\flat}^a$</td>
</tr>
<tr>
<td>S4</td>
<td>$f \cdot \check{\flat}^b = \check{\flat}^a \cdot (f \star f)$</td>
</tr>
</tbody>
</table>

VI. The enzymatic rule

ENZ$_{pa-1}$: if for $f : a + c \rightarrow b + c$ and $g : a + d \rightarrow b + d$ there exists a term $y : c \rightarrow d$ written with $\star$, $\cdot$, $X$, $\check{\flat}$, $\nabla$ (converse of a function)
such that $f \cdot (l_0 \cdot y) = (l_0 \cdot y) \cdot g$, then $f \uparrow ^c = g \uparrow ^d$

Following Milner, one may call the axioms (A) “static laws”. The critical axioms S1–S4 describe the dynamic part of the model with the possibility to copy or delete some components. (Z) is an invariance law which allows using S1–S4 in a cyclic environment.

The kernel of the axioms are the BNA axioms (the resulting algebraic structure is called ax-flow algebra)

B1–B10, R1–R5, and F1–F2,

which form the first two groups of axioms in Table 1. BNA gives a correct and complete axiomatization for flow graphs which are equal modulo graph isomorphism, see, e.g., Chapter 5 in [35]. The remaining graph isomorphism axioms A1–A19 and F3–F5 give a complete axiomatization for the branching constants considered as angelic finite relations, see, e.g., Chapters 6 in [35]. In this case of angelic nondeterminism the divergence is not dominant. E.g. in axioms A3 and A7 the presence of an input–output disconnected path in a branching point does not affect a input–output connected paths. More on the angelic vs. demonic behavior of the branching constants may be found in [3].

This standard version of the calculus of flownomials was designed to handle sequential flowchart algorithms. One goal of the present paper is to study axiomatizations for the branching structures of the data flow networks, starting with the axiomatization of the angelic theory of relations presented in Table 1.

---

3 They are sometimes known in computer science as the “referential transparency” and “garbage collection” properties.
Once the graph isomorphism axioms are already present, one has to add a few very simple axioms (as in \text{S} and \text{Z} above) in order to obtain the classical settings of algebraic theories, matrix theories, iteration theories, as well as regular algebras.

One resulting algebraic structure that is of central interest for our study of deterministic data flow computation is the $d\alpha$-\textit{flow algebra} defined by

- the graph isomorphism axioms with the branching constants $\bot, \land, \top$,
- the critical axioms \text{S3–S4},
- the enzymatic axiom for converses of functions, i.e. for terms only written with $\star, \cdot, I, X, \bot, \land$.

This algebraic structure is dual to the \textit{strong iteration theory} structure of \cite{33} and it is complete for (abstract) flow graphs modulo unfolding equivalence, cf. \cite{33}; see also Theorem 3.3 below or Chapter 8 of \cite{33} for more details.

\textbf{Example 2.1.} As a running example we use the data flow networks graphically presented in Fig. 2(a)–(c). They may be represented by flynomial expressions as well. For instance, the data flow network shown in (a) may be represented by the following expression:

$$\land^a \cdot ![ (l_0 * f \cdot \#X^a) \cdot (f \star I_0) \cdot (\land^a \cdot (f^f \uparrow^a \star I_0) \star^a X^a) ] \uparrow^a .$$

To simplify the writing of such flynomial formulas, we use the convention that feedback has the greatest binding power, then sequential composition, then parallel composition. As we will see below all the networks in Fig. 2 compute the same stream processing function – provided the ramification constant $\land$ is interpreted as the copy constant and the cells are considered deterministic components. Moreover, we show that their equality may be proved using the $d\alpha$-flow axioms. In (d) we have drawn the common infinite tree obtained by unfolding the networks in (a)–(c). In that picture $f_1$
(resp. \(f_2\)) represents the cell obtained from \(f\) by restriction to the first (resp. second) output only.

In the next section we adapt the algebra of cantorian flownomials to data flow nets, i.e. to the cartesian case.

### 3. Deterministic networks

In this section we construct a semantic model SPF\((M)\) for the interpretation of deterministic data flow networks. It is based on stream processing functions.

A stream represents a communication history of a channel. A stream of messages over a given message set \(D\) is a finite or infinite sequence of messages. We define the set of streams \(M\) as

\[
M = D^\omega \quad \text{where} \quad D^\omega = \text{def} \; D^* \cup D^\infty.
\]

By \(x \cdot y\) we denote the result of concatenating two streams \(x\) and \(y\). We assume that \(x \cdot y = x\), if \(x\) is infinite. By \(\langle \rangle\) we denote the empty stream.

If a stream \(x\) is a prefix of a stream \(y\), we write \(x \subseteq y\). The relation \(\subseteq\) is called prefix order. It is formally specified as follows:

\[
x \subseteq y = \text{def} \; \exists z \in M : x \cdot z = y.
\]

The behavior of deterministic interactive systems with \(m\) input channels and \(n\) output channels is modeled by functions

\[
f : M^m \to M^n,
\]

called \((m, n)\)-ary stream processing functions. A stream processing function is called prefix monotonic, if for all tuples of streams \(x, y \in M^m\) we have

\[
x \subseteq y \Rightarrow f(x) \subseteq f(y).
\]

This particular ordering is extended to tuples and functions point-wise in a straightforward way.

A stream processing function \(f\) is called continuous, if \(f\) is monotonic and for every directed set \(S \subseteq M\) we have:

\[
f(\bigcup S) = \bigcup \{ f(x) : x \in S \}.
\]

By \(\bigcup S\) we denote the least upper bound of a set \(S\), if it exists. A set \(S\) is called directed, if for any pair of elements \(x\) and \(y\) in \(S\) there exists an upper bound in \(S\). The set of streams is complete in the sense that for every directed set of streams there exists a least upper bound.

In the following we will use an extension of this setting to the many-sorted case. Let \(S\) be a set of sorts. Let \(D = \{D_s\}_{s \in S}\) be an \(S\)-sorted set of messages and \(M_s = D^*_s \cup D_s^\infty\)
be the set of streams over $D_s$ representing communication histories of channels of type $s$. Concatenation in $S^*$ is denoted by $\cdot$. For $a \in S^*$, $a = a_1 + \cdots + a_{|a|}$, where $a_i$ is the $i$th letter of $a$. Denote by $M_a$ the product $M_{a_1} \times \cdots \times M_{a_{|a|}}$.

Given an $S$-sorted set $D$ of messages and the corresponding sets of streams $M_s$ for $s \in S$, we define the set of stream processing functions with input sorts $a \in S^*$ and output sorts $b \in S^*$ by

$$\text{SPF}(M)(a, b) = \{ f : M_a \rightarrow M_b \mid f \text{ is prefix continuous} \}.$$ 

The BNA constants and operations are interpreted as follows:

- **Parallel composition**: For $f \in \text{SPF}(M)(a, b)$ and $g \in \text{SPF}(M)(c, d)$ the parallel sum $f \star g \in \text{SPF}(M)(a + c, b + d)$ is defined by

  $$(f \star g)(x, y) = (f(x), g(y)) \quad \text{for } x \in M_a \text{ and } y \in M_c.$$ 

- **Sequential composition**: For $f \in \text{SPF}(M)(a, b)$ and $g \in \text{SPF}(M)(b, c)$ the functional composition $f \cdot g \in \text{SPF}(M)(a, c)$ is the usual one defined by

  $$(f \cdot g)(x) = g(f(x)) \quad \text{for } x \in M_a.$$ 

Note that we have used the diagrammatic order.

- **Feedback**: For $f \in \text{SPF}(M)(a + c, b + c)$ the feedback $f \uparrow^c \in \text{SPF}(M)(a, b)$ is defined as follows: For streams $x \in M_a$, we specify

$$f \uparrow^c(x) = \bigsqcup_{k \geq 1} y_k,$$

where the streams $y_k \in M_b$ and $z_k \in M_c$ are inductively defined by

$$(y_1, z_1) = f(x, \langle \rangle) \quad \text{(recall that } \langle \rangle \text{ denotes the empty stream) and}$$

$$(y_{k+1}, z_{k+1}) = f(x, z_k) \quad \text{for } k \geq 1.$$ 

Since $f$ is continuous we can equivalently define $f \uparrow^c$ by fix-point techniques, because

$$(y, z) = \left( \bigsqcup_{k \geq 1} y_k, \bigsqcup_{k \geq 1} z_k \right)$$

is the least fix-point of the function

$$\lambda y, z : f(x, z)$$

and, in other words, the least solution for $y, z$ of the equation

$$(y, z) = f(x, z)$$

\footnote{\(f\) is continuous, hence this definition is well formed.}
• (Block) Identity: \( I_a \in SPF(M)(a,a) \) is defined by
\[
I_a(x) = x, \quad \text{for all } x \in M_a
\]

• (Block) Transposition: \( aX^b \in SPF(M)(a+b,b+a) \) is defined by
\[
aX^b(x,y) = (y,x) \quad \text{for } x \in M_a \text{ and } y \in M_b.
\]

Now we look at the meaning of the various branching constants. In the case of deterministic stream processing functions the meaning of the ramification constants \( \land \) and \( \bot \) is more or less standard: \( \land \) is the copy constant \( \text{\textbackslash r} \) and \( \bot \) is the sink constant \( \text{\textlangle\textrangle} \).

They are defined as follows:

• (Block) Copy: \( \text{\textbackslash r}^a \in SPF(M)(a,a+a) \) is defined by
\[
\text{\textbackslash r}^a(x) = (x,x), \quad \text{for } x \in M_a.
\]

• (Block rich) Sink: \( \text{\textlangle\textcdot\textrangle}^a \in SPF(M)(a,0) \) is defined by
\[
\text{\textlangle\textcdot\textrangle}^a(x) = (\text{\textlangle\textrangle}a), \quad \text{for } x \in M_a,
\]

where \( \text{\textlangle\textrangle} \) denotes the empty tuple of streams.

The constant \( \top \) may be interpreted as a dummy source \( \text{\textlangle\textstar\textrangle} \), defined as follows:

• (Block dummy) Source: \( \text{\textlangle\textstar\textrangle}^a \in SPF(M)(0,a) \) is defined by
\[
\text{\textlangle\textstar\textrangle}^a() = (\text{\textlangle\textrangle}a),
\]

where \( \text{\textlangle\textrangle} \) is the \( a \)-tuple of empty streams \( \text{\textlangle\textrangle} \).

Finally, the constant \( \lor \) is usually left uninterpreted in this case of deterministic data flow networks. Its standard meaning in asynchronous data flow is as the nondeterministic “merge” constant.\(^5\)

3.1. Graph isomorphism

With the operators introduced above, \( SPF(M) \) forms a heterogeneous algebra. This algebra fulfills the BNA axioms.

**Theorem 3.1** (Graph isomorphism). \((SPF(M), \star, \cdot, \uparrow, I_a, aX^b)\) is a BNA model.

**Theorem 3.2** (Graph isomorphism with constants \( \text{\textbackslash r}, \text{\textlangle\textcdot\textrangle}, \text{\textlangle\textstar\textrangle} \)). \((SPF(M), \star, \cdot, \uparrow, I_a, aX^b, \text{\textbackslash r}, \text{\textlangle\textcdot\textrangle}, \text{\textlangle\textstar\textrangle})\) obeys the following additional axioms A5–A9, A12–A13, A16–A19 and F4 in Table 1 with \( \text{\textbackslash r}, \text{\textlangle\textcdot\textrangle}, \text{\textlangle\textstar\textrangle} \) instead of \( \land, \bot, \top \), respectively.

In the terminology of [35] this means that \( SPF \) is a \( d\beta \)-ssmc with feedback. The additional axioms for the branching constants are presented in Table 2, III and IV.

\(^5\) However, in [2] an “equality test” meaning is assigned to the \( \lor \) constant as a “dual” version of the copy constant.
The proof of these theorems is given in detail in Section 5. The main ideas are presented below.

**Sketch of proofs:** It is easy to see that all the axioms without the feedback operator hold. Actually, SPF($M$) is a sub-theory of the algebraic theory Pow($M$) of all the functions on $M$ (defined in [37], for instance) and it is well known that the axioms B1–B10, A5–A8, and A16–A19 in Table 1 are valid in an algebraic theory. (Actually, S3–S4 are valid, too and together with B1–B10, A5–A8, and A16–A19 they give an equivalent axiomatization for algebraic theories.) In addition, A9 and A12–A13 clearly hold. It remains to be shown that the axioms involving the feedback operation are valid, i.e. R1–R5, F1–F2 and F4 in Table 1. The proofs are fairly easy. The most “difficult” proof is that of axiom R5, which shows that a simultaneous multiple feedback is equivalent to repeated unary feedbacks.

3.2. Input–output behavior

The role of this subsection is to present a correct and complete axiomatization for deterministic networks with respect to their equivalence given by the input–output behavior.

It is suggestive to take into account some equivalences on networks which are coarser than graph-isomorphism. Here we consider the equivalence which identify the networks that compute the same input–output function. Two networks which compute the same input–output function are also called SPF-equivalent.

We use the following plan for the proof of the correctness and completeness theorem. (1) The key result we are starting with is the axiomatization of the unfolding equivalence for abstract graphs given by Ştefănescu [33,35]. It is used here in the dual version presented in Theorem 3.3 below, namely as consisting of the $d\times$-flow axioms. (2) Next, we prove the correctness part. To this end, we check the validity of the $d\times$-flow axioms in SPF. By a general result in [35] (Example 4.2.5, p. 54), the $d\times$-flow axioms imply the fixpoint identity, hence this identity is valid in SPF, too. This shows the network unfolding procedure is correct, i.e. it preserves the SPF semantics. (3) For the completeness part, we show that if two networks have the same SPF semantics, then they are unfolding equivalent. (4) Combined with the correctness part, this latter result shows the input–output equivalence on deterministic networks coincides with the unfolding equivalence, hence the $d\times$-flow axioms give an axiomatization for such networks modulo SPF-equivalence. (5) The proof of Theorem 3.3 is given in detail in [35] in a very general, abstract setting. This proof cannot be sketched here. However, to give an impression on the use of the $d\times$-flow axioms we have included an extended example of network transformations in this axiomatic system.

Let us now pass to the technical part. We start with the following result which follows by duality from the similar result proved in [33] or Chapter 8 of [35].
Theorem 3.3 (axiomatization of the unfolding equivalence). The axioms of $d_\alpha$-flow are correct and complete for deterministic dataflow networks modulo unfolding equivalence.\(^6\)

Next, we show the correctness of the unfolding procedure with respect to the SPF semantics.

Definition 3.4. We say that two networks $F$ and $G$ are input–output (or SPF-) equivalent, and write $F \equiv_{IO} G$, iff they have the same SPF-semantics in every interpretation. This means, for all interpretations of the cells as stream processing functions the networks compute the same stream processing function.

The theorems in the previous subsection show that the graph-isomorphism axioms in Table 1 hold in SPF, with respect to the branching constants $\times_{\bowtie}, \cdot$. We check whether some other axioms in Table 1 hold in SPF. First we notice that the strong axioms S3–S4 of Table 1 hold true, hence SPF is an algebraic theory, cf. Chapter 3 in [35]. Then, we find the validity of a particular form of the enzymatic rule. These facts are stated in the following lemma which is proved in Section 5, too.

Lemma 3.5. (1) The strong axioms S3–S4 of Table 1 hold in SPF.

(2) $ENZ_{F_{n-1}}$ holds in SPF.

From (1) of this lemma, the graph-isomorphism axioms, and the dual version of Proposition 3.2.2 in [35] it follows that SPF is an algebraic theory, hence each multiple-output function in SPF is equal to a tuple of one-output functions. Therefore the replacement of a multiple-output cell of a network by a tuple of one-output cells preserves the SPF semantics. Hence we may suppose each cell in a network and the network itself has exactly one output.

A network as above (with one output and such that each cell has one output, too) may be unfolded into a tree. The unfolding of a multiple output network is by definition equal to the tuple of the unfoldings corresponding to each output.

The unfolding procedure is the standard one: it starts with a copy of the output cell of the network which gives the 1st level of the tree; at the 2nd level we put copies of the network cells whose outputs are inputs for the cell of the 1st level; and so on; moreover a proper variable is used for each input channel and it is used whenever is necessary as a terminal vertex in the tree. An example of such an unfolding process is given in Fig. 2 (d). More precisely, in Fig. 2 (d) the first component of the unfolding of the networks in (a)–(c) of the same figure is presented. We want to point out that this kind of unfolding may lead to infinite trees. E.g., the tree in Fig. 1 (d) is infinite on the top.

\(^6\) Recall that this means: (1) the graph isomorphism axioms in Theorem 3.2, i.e., B1–B10, A5–A9, A12–A13, A16–A19, R1–R5, F1–F2, and F4, (2) the strong axioms S3–S4, and (3) the enzymatic axiom for converses of functions, i.e. $ENZ_{F_{n-1}}$.\[^{11}11\]
**Definition 3.6.** We say that two deterministic dataflow networks $F$ and $G$ are *unfolding equivalent*, and write $F \equiv_{\text{unfold}} G$, iff both networks $F$ and $G$ unfold into the same tuple of trees.

Next, we prove the correctness of the unfolding procedure with respect to the SPF semantics.

**Lemma 3.7** (unfolding equivalence $\Rightarrow$ input–output equivalence). If $F' \equiv_{\text{unfold}} F''$, then $F' \equiv_{\text{IO}} F''$.

**Proof.** By a general result (see Example 4.2.4 in [35]), the strong axioms S3–S4 together with the graph isomorphism axioms imply the fixpoint equation, i.e. for $f : a + b \rightarrow b$

$$(f \cdot \bar{R}^b) \uparrow^b = \bar{R}^a \cdot (I_a \star (f \cdot \bar{R}^b) \uparrow^b) \cdot f.$$

Hence, if we unfold a loop once, then an SPF-equivalent network is obtained. Iterating this result one gets the correctness of an arbitrary, but finite, number of loop unfoldings. Finally, the continuity assumption on stream processing functions implies the correctness of the infinite unfolding process. $\square$

Next, we show the converse that two deterministic dataflow networks unfold into the same tuple of trees provided they compute the same input–output function for all functional interpretations of the atomic cells.\(^7\)

**Lemma 3.8** (input–output equivalence $\Rightarrow$ unfolding equivalence). If $F' \equiv_{\text{IO}} F''$ for all functional interpretations of the atoms, then $F' \equiv_{\text{unfold}} F''$.

**Proof.** We give the proof in the one-sorted case only. The proof in the general case is similar. Moreover, instead of the given implication we show the validity of the following equivalent statement: If $F'$ and $F''$ are different trees, then there exists a functional interpretation of the atoms such that $F'$ and $F''$ compute different functions.

Let $D$ be a domain of data consisting of partial $\Sigma$-terms over $X$ (“partial” means that terms $\sigma(x_1, \ldots, x_n)$ with some undefined arguments are allowed; such undefined elements are denoted by “$?$”), where
- $X$ is an infinite set of variables and
- $\Sigma$ is a signature containing an $m$-ary symbol $\sigma_f$ for each atom $f$ with $m$ inputs (and one output) which occurs either in $F'$ or $F''$.

---

\(^7\)This is not the case for the flowchart interpretation of flowgraphs. In that case the unfolding equivalence has to be combined with the reductions of the subtrees without outputs to the empty tree in order to capture the input–output equivalence, see Chapter 10 of [35].
The interpretation is

**Case** \( m \geq 1 \): A cell \( f : m \to 1 \) acts by
\[
f(v_1, \ldots, v_m) = \sigma_f(?, \ldots, ?) \gamma g(v_1, \ldots, v_m),
\]
\[
g(t_1, \ldots, t_m) = \sigma_f(t_1, \ldots, t_m) \gamma g(w_1, \ldots, w_m),
\]
where \( t_1, \ldots, t_m \in D \) and \( v_1, \ldots, v_m, w_1, \ldots, w_m \in D^0 \).

(This definition works for finite streams, too. According to the definition the components of the output stream are defined up to the maximal length of the input streams; thereafter the output is empty.)

**Case** \( m = 0 \): A cell \( f : 0 \to 1 \) produces the output \((\sigma_f)^\infty\), or formally
\[
f() = \sigma_f(?)^\infty.
\]

Take a distinguished variable \( x_i \) for each input \( i \) and consider as input the streams
\[
((x_1)^\infty, \ldots, (x_n)^\infty).
\]

The output
\[
|F|((x_1)^\infty, \ldots, (x_n)^\infty)
\]
produced by a tree \( F : n \to 1 \) is a stream
\[
t_1, t_2, \ldots,
\]
where \( t_i \) is the partial output of \( F \) up to level \( i \).

Since \( F' \) and \( F'' \) are different, there is a level \( i \) such that they are different at level \( i \), hence
\[
|F'|((x_1)^\infty, \ldots, (x_n)^\infty) \neq |F''|((x_1)^\infty, \ldots, (x_n)^\infty)
\]
and the implication is proved. \( \square \)

We illustrate the proof by an example.

**Example 3.9.** The idea of the above proof may be illustrated by the tree in Fig. 2(d) as follows. Under the displayed interpretation, the output computed by the tree network is
\[
\sigma_{f_1}(?,?)^\gamma
\]
\[
\sigma_{f_1}(x_1, \sigma_{f_1}(?,?))^\gamma
\]
\[
\sigma_{f_1}(x_1, \sigma_{f_1}(x_1, \sigma_{f_1}(?,?)))^\gamma
\]
\[
\sigma_{f_1}(x_1, \sigma_{f_1}(x_1, \sigma_{f_1}(x_1, \sigma_{f_1}(?,?))))^\gamma \ldots.
\]

One may see that the first output data gives the approximation of the tree up to level 1, the second up to level 2, and so on.

We collect the above two lemmas in a proposition.
Proposition 3.10 (unfolding equivalence = input–output equivalence). \( F' \equiv \text{unfold} F'' \) iff \( F' \equiv_{IO} F'' \) for all functional interpretations of the atoms.

From this proposition and Theorem 3.3 we get the main result of this subsection.

Theorem 3.11 (axiomatization of the input–output behavior). The \( d\alpha \)-flow axioms give a correct and complete axiomatization for the stream processing functions obtained as interpretations of deterministic dataflow networks. Fully written, the \( d\alpha \)-flow axioms are the BNA axioms I and II in Table 1 and the additional axioms in Table 2.

Example 3.12. The essence of the above theorem may be illustrated with the help of the data flow networks in Fig. 2. One may easily see that the data flow networks drawn in (a)–(c) have the same unfolding. Actually, the unfolding is the pair \((t, t)\), where \( t \) is the tree in Fig. 2(d). \((f_1\) and \(f_2\) in \((d)\) denote the first and the second output component of the cell \( f \) which occurs in \((a)-(c)\).

Let us see how we may prove their equality in the axiomatic system given by the \( d\alpha \)-flow axioms. Besides the graph isomorphism axioms this axiomatic system has two new ingredients: the critical axioms S3–S4 and the invariance/enzymatic axiom \( \text{ENZ}_{F_n^{-1}} \), for the class \( E \) of terms ("enzymes") specified using the branching constants \( \otimes \) and \( \odot \) and the (acyclic) BNA signature.

Of these new axioms, \( \text{ENZ}_{F_n^{-1}} \) is, by far, the most complicated. It may be explained using the following representation of the networks by system of equations. The functions computed by the network in (a)–(c) may also be specified as the least fixpoint solutions corresponding to \( y_1 \) and \( y_2 \) of the systems (N1)–(N3), respectively, where

\[
\begin{align*}
\text{var} &::= x_1 : \text{in}; \quad y_1, y_2 : \text{out}; \quad u, v, z, t : \text{local in} \quad \text{in} \\
v &= y_1, \quad f_1(x_1, t) = y_1, \quad f_1(x_1, w) = y_1, \\
f_1(x_1, z) &= y_2, \quad f_1(x_1, z) = y_2, \quad f_1(x_1, w) = y_2, \\
f_2(x_1, t) &= v, \quad f_2(x_1, t) = z, \quad f_2(x_1, w) = w, \\
f_2(x_1, z) &= t, \\
f_1'(v, u) &= u, \\
\end{align*}
\]

(N1) \quad \quad \quad \quad (N2) \quad \quad \quad \quad (N3),

where \( f_1 = f \cdot (l_1 \star l_1) \) and \( f_2 = f \cdot (l_1 \star l_1) \).

First we consider the enzymatic axiom applied for the set \( E \) of converses of finite injective functions \( \text{In}^{-1} \) considered as abstract relations generated by the acyclic BNA signature and the \( \odot \) constant. In such a system, this rule allows deleting some equations of the system, provided they define variables that are not used in the generation of the output. In the running example \( u \) and the corresponding equation may be deleted as it
does not affect the value of \( y_1 \) or \( y_2 \). Formally, if we write the left-hand-side terms of the system as the tuple

\[
F_1 := (v, f_1(x_1, z), f_1(x_1, t), f_2(x_1, t), f_2(x_1, z), f'(v, u)),
\]

the network behavior is the fixpoint solution corresponding to variables \((y_1, y_2)\) of the equation

\[
F_1(y_1, y_2, v, z, t, u) = (y_1, y_2, v, z, t, u).
\]

By axiom S3,

\[
F_1 \cdot [l_2 \star (l_3 \star \phi^5)] = (v, f_1(x_1, z), f_1(x_1, t), f_2(x_1, t), f_2(x_1, z), \phi^1)
\]

\[
= [l_1 \star (l_3 \star \phi^1)] \cdot F',
\]

where \( F' \) is the tuple

\[
F' := (v, f_1(x_1, z), f_1(x_1, t), f_2(x_1, t), f_2(x_1, z)).
\]

Using the enzymatic rule for enzyme \((l_3 \star \phi^1)\), it follows that the systems specified by \( F_1 \) and \( F' \) have the same solution restricted to variables \((y_1, y_2)\).

Using graph isomorphism transformations the system specified by \( F' \) and restricted to variables \((y_1, y_2)\) is equivalent to the system in (N2) restricted to the variables \((y_1, y_2)\). Notice that (N2) is specified by the tuple

\[
F_2 := (f_1(x_1, t), f_1(x_1, z), f_2(x_1, t), f_2(x_1, z)).
\]

It is a bit more complicated to find the meaning of the enzymatic axiom for converse of surjective functions \( \text{Sur}^{-1} \) considered as abstract relations induced by the acyclic BNA signature and the \( \mathcal{R} \) constant. In this case, we may identify certain variables provided the left-hand side terms in the corresponding equations become equal after the identification of the variables. In the running example, we may identify \( z \) and \( t \), and rewrite them as a new variable \( w \), since after identification the corresponding terms in the system \( f_2(x_1, t) \) and \( f_2(x_1, z) \), become equal, namely both become equal to \( f_2(x_1, w) \). Formally, by axiom S4 we get

\[
[l_2 \star \mathcal{R}^1] \cdot F_2 = (f_1(x_1, w), f_1(x_1, w), f_2(x_1, w), f_2(x_1, w)) =_{S4} F_3 \cdot [l_2 \star \mathcal{R}^1],
\]

where

\[
F_3 := (f_1(x_1, w), f_1(x_1, w), f_2(x_1, w)).
\]

Using the enzymatic rule for enzyme \( \mathcal{R}^1 \), it follows that the systems specified by \( F_2 \) and \( F_3 \) have the same solution restricted to variables \((y_1, y_2)\). The resulting system specified by \( F_3 \) is shown in (N3).

Finally, we observe that: (1) each network/system may be minimized using transformations as in the example above, (2) minimal networks are unique up to isomorphism
and (3) minimal networks are in bijective correspondence with the unfolding trees. Using this latter result, in the above example we have used minimal networks/systems with respect to the enzymatic transformations above, rather than unfolding trees. The completeness proof for the unfolding equivalence is done via minimal networks using the technique shown in the above example. The proof is long and it is based on many other results and cannot be sketched here; see Chapter 8 in [35] for a detailed proof.

Following [30], we say a stream processing function is a \( \{\mathcal{R}, \phi, \star\} \)-buffering morphism if it is specified by a data flow network built up with the BNA operations and

- the copy \( \mathcal{R} \), sink \( \phi \), and source \( \uparrow \) branching constants and,
- the trivial cells \( s_d \in \text{SPF}(0, s) \) for \( d \in M_\mathcal{r} \), \( s \in S \) defined by \( s_d(\cdot) = d \).

Such buffering morphisms not only duplicate, lose, or route the input data, but also may insert particular data via their trivial cells.

4. Nondeterministic networks

Deterministic data flow nets have more or less a canonical denotational semantics, which was defined and used in the previous section. To find such a semantics for nondeterministic networks is less obvious. The semantics of nondeterministic data flow networks may be reduced to the semantics of deterministic networks using oracles. Such an oracle fixes a priori the behavior of the network regarding the nondeterministic points. Given a fixed oracle, a nondeterministic network becomes deterministic and it computes a stream processing function. Varying the oracle we obtain the semantics of a nondeterministic network as a set of stream processing functions.

Formally, we construct the model \( \mathcal{PSPF}(M) \) for the interpretation of nondeterministic data flow networks as follows.

First, for streams \( a, b \in S^{\star} \) define

\[
\mathcal{PSPF}(M)(a,b) := \{ F \mid F \subseteq \text{SPF}(a,b) \}.
\]

Next, the operations \( \star, \cdot, \uparrow \) are defined in an elementwise manner by

\[
F \star G = \{ f \star g \mid f \in F, g \in G \},
\]

\[
F \cdot G = \{ f \cdot g \mid f \in F, g \in G \},
\]

\[
F \uparrow = \{ f \uparrow \mid f \in F \}.
\]

Then, each constant \( c \in \{ I, X, \mathcal{R}, \phi, \star\} \) of SPF induces a corresponding constant \( \{ c \} \) of \( \mathcal{PSPF} \).

In this model, we may give meaning to additional nondeterministic branching constants, namely,

- (Block) Split: for \( a \in S \)

\[
\mathcal{R}^a = \{ \phi^a \mid \phi : \omega \rightarrow \{ 1, 2 \} \},
\]
where for an oracle \( \phi \), 
\[
\phi^a(x) = \text{def} (y; z),
\]
with \( y \) and \( z \) obtained by splitting \( x \) according to \( \phi \). That is, if \( \phi(i) = 1 \) then the \( i \)th input is forwarded on output channel 1, otherwise on output channel 2. This definition is extended to arbitrary words \( a \in S^* \) using the identities in A18–A19.8

• **(Block) Merge**:9 for \( a \in S^* \)

\[
\forall_a = \{ \phi_a \mid \phi : \omega \rightarrow \{1, 2\} \},
\]

where for an oracle \( \phi \), 
\[
\phi^a(x; y) = z
\]
with \( z \) obtained from \( x \) and \( y \) according to \( \phi \). This means, the 1st output element in \( z \) is taken from input channel \( \phi(1) \), the second from channel \( \phi(2) \), and so on. If the oracle requires data from an input channel which is empty, then the merge cell is blocked and no more output data are delivered.

With A14–A15 this definition is extended to arbitrary \( a \in S^* \).

• **(Block rich) Source**: for \( a \in S^* \)

\[
\forall_a = \{ g_x \mid x \in M_a \},
\]

where for \( x \in M_a \), \( g_x : 0 \rightarrow a \) is the function given by \( g_x( ) = x \).

By split, merge, and source we have introduced three nondeterministic constants for data flow nodes.10

4.1. Graph isomorphism

As is well known, for nondeterministic terms certain classic algebraic equations do not hold. Examples for such equations are provided by the strong commuting identity S4 or the “fixpoint equation”. Nevertheless, all equations characterizing graph isomorphisms hold, of course.

**Theorem 4.1** (Graph isomorphism). \( (\mathcal{P}^{SPF}, \ast, \cdot, \uparrow, l, x) \) is a BNA model.

**Proof.** The proof follows directly from the corresponding result in the deterministic case, i.e. Theorem 3.1. The key point is the observation that all the BNA axioms11 are identities with both the left- and the right-hand-side terms containing at most one

---

8 Notice that we have independent oracles for each input channel in \( a \) and not a unique one for all the inputs in \( a \). Axiom A19 would fail with a definition that uses the latter version.

9 We define here a merge that is neither nonstrict nor fair. The treatment of a fair and nonstrict merge needs a more sophisticated semantic model (see [11]).

10 A nondeterministic sink node, dual to the source node above, cannot be defined in this denotational model \( \mathcal{P}^{SPF} \). Such a node simply drops the input! However, at an operational level one may distinguish between consuming all input data and producing nothing, while the node rejects to consume input data. See [2] for more on these operational definitions.

11 Recall, the BNA axioms are B1–B10, R1–R5 and F1–F2 in Table 1.
occurrence of a variable and each variable that occurs in one part of an identity occurs in the other part, as well. Hence the validity of the proof of a BNA axiom in $\mathcal{SPF}$ may be checked on elements and it follows from the validity of the corresponding axiom in SPF. □

To these basic axioms for graph isomorphism we can add equations for the branching constants interpreted as split-merge cells.

**Theorem 4.2** (Graph isomorphism with various constants). $(\mathcal{SPF}, \ast, \cdot, \uparrow, \trianglerightleftarrow, 1, X, \wedge, \vee, \uparrow, \downarrow)$ obeys the additional axioms $A1\text{--}A2$, $A4\text{--}A6$, $A8\text{--}A9$, $A12\text{--}A19$ and $F3\text{--}F4$ in Table 1 where $\wedge, \nabla, \trianglerightleftarrow$ replace $\land, \bot, \lor, \top$, respectively. For the remaining axioms, only one inclusion holds, i.e. “$\subset$” for A3, A7, A11 and F5 and “$\supset$” for A10 do not hold. □

For convenience, the resulting axioms are collected in Table 3. The details of the proof may be found in Section 5. Since axioms A1–A2 and A5–A6 are valid, the oracle-based semantics of the nondeterminism is associative and commutative. Hence we may equivalently use the extended branching constants

\[
\forall k : a \rightarrow ka \quad \text{and} \quad \forall a : ka \rightarrow a \quad \text{for } k \geq 1,
\]

where $\phi : \omega \rightarrow \{1, \ldots, k\}$ is a $k$-oracle. On the other hand, axioms A3 and A7 do not hold, hence we have a nonfair merge and therefore a nonangelic calculus of relations modeling the split-merge branching structure of nondeterministic networks.
4.2. The input–output behavior

In this section we look at the axiomatization problem for the input–output behavior of nondeterministic data flow networks.

It is easy to see that neither the strong axioms S1–S2 nor S3–S4 in Table 1 hold. Similarly, due to the nondeterministic behavior of the cells the fixpoint identity is not valid, hence the unfolding of networks is not a correct rule. All these comments amount to say that algebraic or iteration theories cannot be used in this setting.

These observations lead towards a counterexample to a thesis\textsuperscript{12} of Bloom and Esik: Whenever an iterative process is present an iteration theory structure may be found (see, e.g. [4]).

There are many examples which were studied in full detail by Bloom and Esik showing that this is the case when one tries to capture the iteration laws in combination with the algebraic theory laws. On the other hand, SPF provides an example of a natural iterative process which is neither an algebraic theory nor a dual algebraic theory (i.e., neither S1–S2 nor S3–S4 of Table 1 hold). Since an iteration theory is an algebraic theory we get the following result:

The thesis of Bloom and Esik is false.

By contrast, the flownomial calculus starts with an axiomatization of the iteration operation combined with the monoidal category primitives rather than with the algebraic theory primitives. This is the key reason for the successful application of the flownomial calculus to the case of nondeterministic data flow networks, as it has been presented in the previous subsection.

One may perhaps suggest to replace the Bloom and Esik thesis above by the following weaker one:

Whenever an iterative process is present the BNA laws hold, hence an ax-flow algebra may be found.

Since the BNA laws are correct and complete for graphs modulo graph isomorphism, this is true whenever one is able to find a compositional graphical description of the underlying iterative process.

The problem of axiomatizing the input–output behavior of nondeterministic data flow networks is still open even in the case when only branching constants are present. To be more precise, let us define a \{\texttt{\&\&}, \texttt{\lor}, \texttt{\&}, \texttt{\lor}, \texttt{\&}, \texttt{\lor}, \texttt{\&}, \texttt{\lor}, \texttt{\&}\}-buffering morphism as a set of stream processing functions specified by a data flow network built up with

- the split \texttt{\&}, merge \texttt{\lor}, (rich) sink \texttt{\&}, (dummy) source \texttt{\lor}, (rich) source \texttt{\&} and copy \texttt{\&} constants
- the trivial cells \texttt{sd} \in SPF(0,s) for \( d \in M(s \in S) \) defined by \( sd() = d \).

Let \( R \) be a subset of branching constants in \{\texttt{\&\&}, \texttt{\lor}, \texttt{\&}, \texttt{\lor}, \texttt{\&}, \texttt{\lor}\}. We are interested in the following problems for an arbitrary \( R \) and either for arbitrary networks or for acyclic networks, only.

\textsuperscript{12} Strictly speaking, this is a thesis and not a conjecture since it states that the informal notion of an iterative process is captured by the formal definition of iteration theories.
• **Expressiveness**: Characterize the \( R \)-buffering morphisms. (Certain invariants and/or complexity measures may be useful to classify the equivalent networks.)

• **Decidability**: Check the decidability of the equality problem for various sets \( R \) of branching constants.

• **Axiomatization**: Give complete (and correct) axiomatizations for the \( R \)-buffering morphisms.

  In general, the problem of a complete axiomatization is open although we have certain partial results. To be more specific, the problem seems to be difficult for subsets \( R \) which contain both the split and merge constants.

5. More proofs

In this section we give detailed proofs for those theorems of the previous sections whose proofs were skipped before. Mainly, these are the graph-isomorphism theorems.

**Lemma 5.1** (Axiom R5 of Table 1).

\[
(f \uparrow^{c+d}) = (f \uparrow^d) \uparrow^c
\]

for \( f \in \text{SPF}(M)(a+c+d, b+c+d) \). Hence one application of a multiple feedback may be replaced by repetitive applications of unary feedbacks.

**Proof.** Let \( f \in \text{SPF}(M)(a+c+d, b+c+d) \). Then:

- \( f \uparrow^{c+d} \in \text{SPF}(M)(a,b) \) is defined by
  \[
  (f \uparrow^{c+d})(x) = y,
  \]
  where \( y = \bigcup_{k \geq 1} y_k \) and \( y_k, z_k, w_k \) are inductively defined by
  \[
  (y_k, z_k, w_k) = f(x, z_{k-1}, w_{k-1}) \quad \text{for } k \geq 1,
  \]
  where \( z_0 = \langle \rangle, w_0 = \langle \rangle \).

  Denote
  \[
  z := \bigcup_{k \geq 1} z_k,
  \]
  \[
  w := \bigcup_{k \geq 1} w_k.
  \]

- \( (f \uparrow^d) \uparrow^c \in \text{SPF}(M)(a,b) \) is defined as follows:
  \[
  ((f \uparrow^d) \uparrow^c)(x) = \overline{y},
  \]
  where \( \overline{y} = \bigcup_{i \geq 1} \overline{y}_i \) and \( \overline{y}_i, z_i \) are inductively defined by
  \[
  (\overline{y}_i, z_i) = (f \uparrow^d)(x, z_{i-1}) \quad \text{for } i \geq 1,
  \]
where $z_0 = \langle \rangle$, hence by the definition of $f \uparrow^d$ there are elements $\tilde{y}_{i,j}, \tilde{z}_{i,j}, \tilde{w}_{i,j}$ for $i,j \geq 1$ such that for all $i \geq 1$:

$$\overline{y}_i = \bigcup_{j \geq 1} \tilde{y}_{i,j}, \quad \overline{z}_i = \bigcup_{j \geq 1} \tilde{z}_{i,j}$$

and

$$(\tilde{y}_{i,j}, \tilde{z}_{i,j}, \tilde{w}_{i,j}) = f(x, \overline{z}_{i-1}, \overline{w}_{i-1}) \quad \text{for } j \geq 1,$$

where $\tilde{w}_{i,0} = \langle \rangle$.

It is obvious that each sequence $(\tilde{y}_{i,j})$, $(\tilde{z}_{i,j})$, and $(\tilde{w}_{i,j})$ is increasing on both indices $i,j$, hence the following notation makes sense:

$$\overline{z} := \bigcup_{i \geq 1} z_i$$

and

$$\overline{w} := \bigcup_{i \geq 1} w_i \quad \text{where for } i \geq 1: \overline{w}_i := \bigcup_{j \geq 1} \tilde{w}_{i,j}.$$ 

**Proof of** $f \uparrow^{c+d} = (f \uparrow^d)^\uparrow^c$.

(a) $f \uparrow^{c+d} \sqsubseteq (f \uparrow^d)^\uparrow^c$

First note that

$$(y_k, z_k, w_k) \sqsubseteq (\tilde{y}_{k,k}, \tilde{z}_{k,k}, \tilde{w}_{k,k}), \quad \forall k \geq 1.$$ 

Indeed, for $k = 1$ it follows by

$$(y_1, z_1, w_1) = f(x, z_0, w_0)$$

$$= f(x, \langle \rangle, \langle \rangle)$$

$$= f(x, z_0, \tilde{w}_{1,0})$$

$$= (\tilde{y}_{1,1}, \tilde{z}_{1,1}, \tilde{w}_{1,1})$$

and if it holds for $k$, then it holds for $k + 1$ by

$$(y_{k+1}, z_{k+1}, w_{k+1}) = f(x, z_k, w_k)$$

$$\sqsubseteq f(x, \tilde{z}_{k,k}, \tilde{w}_{k,k})$$

$$\sqsubseteq f(x, z_k, \tilde{w}_{k,k})$$

$$\sqsubseteq f(x, z_k, \tilde{w}_{k+1,k})$$

$$= (\tilde{y}_{k+1,k+1}, \tilde{z}_{k+1,k+1}, \tilde{w}_{k+1,k+1}).$$
By this we get

\[(f \uparrow^{c+d})(x) = y\]

\[= \bigcup_{k \geq 1} y_k\]

\[\subseteq \bigcup_{k \geq 1} \tilde{y}_{k,k}\]

\[= \bigcup_{i \geq 1} \bigcup_{j \geq 1} \tilde{y}_{i,j}\]

\[= \bigcup_{i \geq 1} \tilde{y}_i\]

\[= \tilde{y}\]

\[= ((f \uparrow^{d}) \uparrow^c)(x)\].

(b) \( (f \uparrow^{d}) \uparrow^c \subseteq f \uparrow^{c+d} \)

We prove by a double induction that

\[(\tilde{y}_{i,j}, \tilde{z}_{i,j}, \tilde{w}_{i,j}) \subseteq (y, z, w), \ \forall i, j \geq 1.\]

First note that

\[(y, z, w) = f(x, z, w).\]

Indeed,

\[f(x, z, w) = f\left(x, \bigcup_{k \geq 1} z_k, \bigcup_{k' \geq 1} w_{k'}\right)\]

\[= \bigcup_{k \geq 1} f(x, z_k, w_k)\]

\[= \bigcup_{k \geq 1} (y_{k+1}, z_{k+1}, w_{k+1})\]

\[= (y, z, w).\]

For \(i = 1\): If \(j = 1\) then we have

\[(\tilde{y}_{1,1}, \tilde{z}_{1,1}, \tilde{w}_{1,1}) = f(x, \tilde{z}_0, \tilde{w}_{1,0})\]

\[= f(x, (\}, \langle \})\]

\[= (y_1, z_1, w_1)\]

\[\subseteq (y, z, w)\]
and the passing from $j$ to $j + 1$ follows by

$$(\tilde{y}_{1,j+1}, \tilde{z}_{1,j+1}, \tilde{w}_{1,j+1}) = f(x, \tilde{z}_0, \tilde{w}_{1,j})$$

$$= f(x, \langle \rangle, \tilde{w}_{1,j})$$

$$\sqsubseteq f(x, z, w)$$

$$= (y, z, w).$$

The inductive step from $i$ to $i + 1$: If $j = 1$, then

$$(\tilde{y}_{i+1,1}, \tilde{z}_{i+1,1}, \tilde{w}_{i+1,1}) = f(x, \tilde{z}_i, \tilde{w}_{i+1,0})$$

$$= f \left(x, \bigcup_{j' \geq 1} \tilde{z}_{i,j'}, \langle \rangle \right)$$

$$\sqsubseteq f(x, z, w)$$

$$= (y, z, w)$$

and the passing from $j$ to $j + 1$ is similar as in the previous case $i = 1$. □

Proof (Theorem 3.1). The validity of the axioms without feedback B1–B10 is obvious. R1 may be proved as follows. Let $f: a' \to a$, $g: a + c \to b + c$, $h : b \to b'$ and $x \in M_{a'}$. Then

$$[f \cdot (g \uparrow c) \cdot h](x) = h(y),$$

where $y = \bigsqcup_k y_k$ for $y_k \in M_b$, $z_k \in M_c$ inductively defined by

$$(y_1, z_1) = g(f(x), \langle \rangle, c),$$

$$(y_{k+1}, z_{k+1}) = g(f(x), z_k) \quad (k \geq 1).$$

On the other hand,

$$[(f \star_1) \cdot g \cdot (h \star_1)] \uparrow^c (x) = t,$$

where $t = \bigsqcup_k t_k$ for $t_k \in M_{b'}$, $t_k' \in M_{b'}$, $w_k \in M_c$ inductively defined by

$$t_k = h(t_k') \quad (k \geq 1),$$

$$(t_1', w_1) = g(f(x), \langle \rangle, c)),$$

$$(t_{k+1}', w_{k+1}) = g(f(x), w_k) \quad (k \geq 1).$$

By induction it follows that $h(y_k) = t_k$ and $z_k = w_k$ for all $k$. Hence $h(y) = t$, i.e. R1 is valid. R2 may be proved in a similar way.
For R3 take \( x \in M_\alpha \). Then

\[
[f \cdot (I_b \star g)]^c (x) = y,
\]

where \( y = \bigsqcup_k y_k \) for \( y_k \in M_b \), \( z_k' \in M_d \), \( z_k \in M_c \) inductively defined by

\[
\begin{align*}
z_k &= g(z_k') \quad (k \geq 1), \\
(y_1, z_1') &= f(x, \langle \rangle_c), \\
(y_k+1, z_{k+1}') &= f(x, z_k) \quad (k \geq 1)
\end{align*}
\]

and

\[
[(I_a \star g) \cdot f]^d (x) = t,
\]

where \( t = \bigsqcup_k t_k \) for \( t_k \in M_b \), \( w_k \in M_d \) inductively defined by

\[
\begin{align*}
(t_1, w_1) &= f(x, g(\langle \rangle_d)), \\
(t_{k+1}, w_{k+1}) &= f(x, g(w_k)) \quad (k \geq 1).
\end{align*}
\]

Since \( \langle \rangle_c \sqsubseteq g(\langle \rangle_d) \) we get \( (y_1, z_1') \sqsubseteq (t_1, w_1) \). This implies \( z_1 = g(z_1') \sqsubseteq g(w_1) \), hence \( (y_2, z_2') \sqsubseteq (t_2, w_2) \) and so on. This proves one inclusion

\[
y = \bigsqcup_k y_k \sqsubseteq \bigsqcup_k t_k = t.
\]

For the opposite inclusion, first note that \( z_1 = g(z_1') \sqsupseteq g(\langle \rangle_d) \), hence \( (y_2, z_2') \sqsupseteq (t_1, w_1) \). This implies \( z_2 = g(z_2') \sqsupseteq g(w_1) \), hence \( (y_3, z_3') \sqsupseteq (t_2, w_2) \) and so on. This shows that

\[
y = \bigsqcup_k y_{k+1} \sqsupseteq \bigsqcup_k t_k = t
\]

and R3 is proved.

R4 is obvious, R5 has been proved in Lemma 5.1 and F1, F2 are obviously valid. \( \square \)

**Proof (Theorem 3.2).** The validity of the axioms A5–A9, A12–A13, A16–A19, and F4 of Table 1 with \( \oplus, \odot, \star, \cdot \) instead of \( \wedge, \perp, \top \) is obvious. \( \square \)

**Proof (Lemma 3.5).** S3–S4 are easy to prove. For ENZ-correctness, by hypothesis we have

\[
f \cdot (I_b \star r) = (I_a \star r) \cdot g,
\]

where \( r : M_c \rightarrow M_d \) is such that if \( r(x) = y \), then for each \( j \in \{1, \ldots, |d|\} \) there is a unique \( i \in \{1, \ldots, |c|\} \) such that

(\( * \) \( y_j = x_i \).
The results \( s = (f \uparrow^c)(x) \) and \( u = (g \uparrow^d)(x) \) are determined by the fixpoint iterations:

\[
(s_0, t_0) = (\langle \rangle, \langle \rangle), \quad (u_0, v_0) = (\langle \rangle, \langle \rangle),
\]

\[
(s_{i+1}, t_{i+1}) = f(x, t_i), \quad (u_{i+1}, v_{i+1}) = g(x, v_i).
\]

We obtain

\[
r(t_0) = v_0 \land s_0 = u_0.
\]

(The former equality follows from \((*)\).

Moreover, assuming \( y(t_i) = v_i \land s_i = u_i \) we obtain

\[
(u_{i+1}, v_{i+1}) = g(x, v_i)
\]

\[
= g(x, y(t_i))
\]

\[
= (s_{i+1}, y(t_{i+1})) \quad \text{where} \quad (s_{i+1}, t_{i+1}) = f(x, t_i).
\]

This gives us all we need for an induction proof on \( i \) that shows \( y(t_i) = v_i \land s_i = u_i \).

\( \square \)

**Proof of Theorem 4.2.** First of all, we explain the interplay between the branching constants. The meaning of \( \land \) and \( \lor \) as the split and merge constants, respectively, is taken for granted. In order to have a theory which is closed under the feedback operation, we have to see which is the result of the application of the feedback to such constants.

It is easy to see that

\[
\hat{\phi}^s \uparrow^c = \hat{s}
\]

for all oracles \( \phi \), hence \( \hat{\phi}^s \uparrow^c = \hat{s} \). This equality reflects the fact that our feedback is the least fixed point solution.

For the other constant one may see that

\[
\hat{\phi}^s \uparrow^c = \hat{s}
\]

for all oracles \( \phi \). (For each oracle \( \phi \), the merge function \( \hat{\phi}^s \) is continuous, hence

\[
\hat{\phi}^s \uparrow^c
\]

is a well-defined function and has to be equal to the unique function \( \hat{\phi}^s : s \to 0 \).

All these amount to saying that every set of branching constants including the split and merge constants and closed to the network algebra operations contains \( \{\hat{\land}, \hat{\lor}, \hat{\phi}, \hat{\star}\} \).

We use extended oracles \( \phi : \omega \to \{1, \ldots, k\} \) for \( k \geq 1 \). For instance, the meaning of such an oracle in the case of the split constant \( \hat{\phi}_{k}^s \) is to show the number of the output channel where the current token is sent to. Similarly for the merge constant.
Axioms A14–A15 and A18–A19 hold by definition. On the other hand, it is easy to see that A12–A13 and A16–A17 hold. Hence we may restrict ourself to the analysis of the remaining axioms in the case of single channels, i.e. $a = s \in S$.

For axiom A1 it is enough to see that both terms are equal to $s \vee s$. Clearly,

$$
\left( \psi_s \star l_s \right) \psi_s = s \vee s,
$$

where $\psi$ is the 3-oracle obtained from $\psi'$ and $\psi''$ according to the left-hand side formula. Similarly for the right-hand-side term. The proof is finished showing that a 3-oracle may be simulated by 2-oracle in both ways corresponding to the left- and right-hand-side term of the identity, respectively.

A5 may be proved in a similar way.

For A2 and A6 it is enough to replace an oracle $\psi : \omega \rightarrow \{1,2\}$ by the oracle $\tilde{\psi}$ obtained interchanging numbers 1 and 2.

Axioms A4 holds since for all oracles $\psi$ one has $s \vee s = s \vee s$.

Axiom A8 holds, too. (The splitting of an empty stream is a couple of empty streams.)

Clearly, $s \vee s = 0$, hence A9 is valid.

Finally, axioms F3 and F4 are valid, as we have already seen in the beginning part of the proof.

In the remaining part of the proof we show that the other axioms do not hold.

For A3, one may see that $[(s \star l_s) \psi_s](x)$ is the prefix of $x$ up to the maximal token $k$ such that $\psi(1) = \cdots = \psi(k) = 2$. Hence A3 is not valid, but the “$\supseteq$” inclusion holds. On the other hand, it is interesting to note that varying $\psi$ and keeping fixed $x$ we get the *prefix closure* of $x$.

For the dual axiom A7, one may see that

$$
\left[ s \star l_s \right](x)
$$

is the sub-stream of $x$ given by those positions $k$ for which $\psi(k) = 2$. Hence A7 fails, but the inclusion “$\supseteq$” holds. In this case, varying $\psi$ and keeping fixed $x$ we get the *sub-stream closure* of $x$.

For A10 one may see that

$$
E(\psi', \psi'', \psi', \psi'') = \left( \psi_s \star \psi_s \right) \left( l_s \star s \star l_s \right) \left( \psi_s \star \psi_s \right)
$$

generate a larger class of stream processing functions than

$$
F(\sigma, \tau) = \psi_s \cdot \psi_s.
$$
Indeed,

\[ E(\phi', \phi'', \psi', \psi'')(1^\infty a^\infty b^\infty \ldots) = (2^\infty b^\infty \ldots) \]

for \( \phi' = 1^\infty \); \( \phi'' = 1^\infty \); \( \psi' = 2^\infty \); \( \psi'' = 1^\infty \). On the other hand, this output is not possible for

\[ F(\sigma, \tau)(1^\infty a^\infty \ldots) \]

since the first output on at least one channel here is in the set \{1, a\}.

Conversely, it may be seen that \( F(\sigma, \tau) \) may be simulated by \( E(\phi', \phi'', \psi', \psi'') \) if one takes \( \phi' \) and \( \phi'' \) as certain restrictions of \( \tau \) and \( \psi' \) and \( \psi'' \) as certain restrictions of \( \sigma \). More precisely, for an oracle \( \alpha \) and a subset of natural numbers \( A \subseteq \omega \) let us denote by \( \alpha|_A \) the oracle obtained by restricting \( \alpha \) to \( A \), i.e. if \( A \) consists of the elements \( a_1 < a_2 < \ldots \) then \( \alpha|_A(i) = \alpha(a_i) \) for \( i = 1, 2, \ldots \). Now

\[ \phi' = \tau|_{\sigma^{-1}(1)}, \quad \phi'' = \tau|_{\sigma^{-1}(2)}, \quad \psi' = \sigma|_{\tau^{-1}(1)}, \quad \psi'' = \sigma|_{\tau^{-1}(2)}. \]

(In case certain oracles as above are definite, we may extend them to infinite oracles in an arbitrary way and the result holds.)

With respect to A11, one may easily see that

\[ E(\phi, \psi) = \bigwedge_{\phi}^\psi \]

generates a set of functions which properly includes \( I_\gamma \).

Finally, the left-hand side of F5 specifies a bag,\(^{13}\) hence the corresponding set of functions properly include \( I_\gamma \). \( \square \)

This concludes our proofs.

6. Conclusion

Although we use the same diagrams for data flow graphs with deterministic data flow nodes as well as for nondeterministic data flow nodes, their algebraic axiomatization is essentially different. Not all the laws that hold in the deterministic case also hold in the nondeterministic case. This demonstrates how helpful semantic models are as guidelines when determining the axioms and their soundness. In the deterministic case, we get a complete axiomatization. This is an open problem for the nondeterministic case, however.

\(^{13}\) A bag is similar to a queue, except for the order. More precisely, a bag is a cell that receives data on its input channel and delivers them on the output channel in an arbitrary order. No data are duplicated or lost. Interestingly, the left-hand-side term in F5 gives an implementation of the bag as a circular queue with one entry and one exit point similar to the way of delivering bags in an airport.
The axiomatization of data flow graphs is not only of theoretical interest, it is also of practical relevance. Many software engineering description techniques and their support tools incorporate data flow diagrams. For them, an axiomatization is useful for the manipulation and transformation of data flow graphs.

Acknowledgements

It is a pleasure to thank Ch. Facchi for help in preparing the manuscript and R. Grosu for stimulating discussions on the algebra of stream processing functions. We are also indebted to the anonymous referees for their suggestions for improving the presentation.

References