Network Algebra for Synchronous and Asynchronous Dataflow

J.A. Bergstra[1,3,†] C.A. Middelburg[2,3] Gh. Ştefănescu[4,‡]

[1] *Programming Research Group, University of Amsterdam P.O. Box 41882, 1009 DB Amsterdam, The Netherlands*

[2] *Department of Network & Service Control, KPN Research P.O. Box 421, 2260 AK Leidschendam, The Netherlands*

[3] *Department of Philosophy, Utrecht University P.O. Box 80126, 3508 TC Utrecht, The Netherlands*

[4] *Institute of Mathematics of the Romanian Academy P.O. Box 1-764, 70700 Bucharest, Romania*

E-mail: `janb@fwi.uva.nl` - `keesm@phil.ruu.nl` - `ghstef@imar.ro`

Abstract

Network algebra is proposed as a uniform algebraic framework for the description and analysis of dataflow networks. An equational theory, called BNA (Basic Network Algebra), is presented. BNA, which is essentially a part of the algebra of flownomials, captures the basic algebraic properties of networks. For synchronous and asynchronous dataflow networks, additional constants and axioms for connections are given; and corresponding process algebra models are introduced. The main difference between these models is in the interpretation of the identity connections, called wires in dataflow networks. The process algebra model for the asynchronous case is compared with previous models.

Keywords & Phrases: dataflow networks, network algebra, process algebra, asynchronous dataflow, synchronous dataflow, feedback, merge anomaly, history models, oracle based models, trace models.

1994 CR Categories: F.1.1, F.1.2, F.3.2., D.1.3., D.3.1.

†The first author has been partially supported by ESPRIT BRA 8533 (NADA) and ESPRIT BRA 6454 (CONFER).
‡The third author has been partially supported by HMC cooperation network ERBCHRXCT930406 (EXPRESS).

Contents

1 Introduction

In this paper we pursue an axiomatic approach to the theory of dataflow networks. Network algebra is presented as a general algebraic setting for the description and analysis of dataflow networks. A network can be any labelled directed hypergraph that represents some kind of flow between the components of a system. For example, flowcharts are networks concerning flow of control and dataflow networks are networks concerning flow of data. Assuming that the components have a fixed number of input and output ports, such networks can be built from their components and (possibly branching) connections using parallel composition ($\mathbin{+\!\!\!+}$), sequential composition ($\circ$) and feedback ($\uparrow$). The connections needed are at least the identity (I) and transposition (X) connections, but branching connections may also be needed for specific classes of networks – e.g. the binary ramification ($\wedge$) and identification ($\vee$) connections and their nullary counterparts ($\perp$ and $\top$) for flowcharts.

An equational theory concerning networks that can be built using the abovementioned operations with only the identity and transposition constants for connections, called BNA (Basic Network Algebra), is presented. The axioms of BNA are sound and complete for such networks modulo graph isomorphism. BNA is the core of network algebra; for the specific classes of networks covered, there are additional constants and/or axioms. Flowcharts constitute one such class. BNA is essentially a part of the algebra of flownomials of Căzănescu and Ştefănescu [23] which was developed for the description and analysis of flowcharts.

In addition to BNA, extensions for synchronous and asynchronous dataflow networks are presented. In both cases, process algebra models are given. These models provide for a very straightforward connection between network algebra and process algebra. Process algebra is closely related to programming, whereas network algebra is used for describing systems as a network of interconnected components. A clear connection between them appears to be useful. The process algebra model of asynchronous dataflow networks is additionally connected with various previous models of these networks, including Kahn's history model [34], Broy's oracle based models [19], and Jonsson's trace model [33].

For the process algebra models, ACP (Algebra of Communicating Processes) of Bergstra and Klop [12] is used, with the silent step and abstraction, as well as the following additional features: renaming, conditionals, iteration, prefixing and communication free merge. Besides, a discrete-time extension of process algebra is used to model synchronous dataflow networks.

There are strong connections between the work presented in this paper and other recent work. SCAs (Synchronous Concurrent Algorithms), introduced by Thompson and Tucker in [44], can be described in the extension of BNA for synchronous dataflow networks. In [10], Barendregt et al. present a model of computable processes which is essentially a model of BNA; but a slightly different choice of primitive operations and constants is used. It is also worth mentioning that the examples of Brock and Ackermann [18] and Russell [40] demonstrating a time anomaly in asynchronous dataflow networks are presented in a concise way in this paper, using network algebra for describing the networks of cells and wires and using process algebra for describing the atomic cells.

The paper starts with an outline of network algebra (Section 2) and some process algebra preliminaries (Section 3). Next the signature, the axioms and two models of BNA, including a general process algebra model, are presented (Section 4). Thereafter the signature, the axioms and the process algebra models of the network algebras for synchronous and asynchronous dataflow are presented (Section 5 and Section 6, respectively). For synchronous dataflow networks, a more abstract model based on stream transformers is given directly (Section 5). For asynchronous dataflow networks, several more abstract models are derived from the process algebra model and compared (Section 7). The non-standard mathematical notation for sets, sequences and tuples used in this paper is explained in an appendix.

2 Overview of network algebra

This section gives an idea of what network algebra is. The meaning of its operations and constants is explained informally making use of a graphical representation of networks. Besides, dataflow networks are presented as a specific class of networks and the further subdivision into synchronous and asynchronous dataflow networks is explained in broad outline. The formal details will be treated in subsequent sections.

2.1 General

In the first place, the meaning of the operations and constants of BNA mentioned in Section 1 ($\mathbin{+\!\!+}$, $\circ$, $\uparrow$, I and X) is explained. Following, the meaning of the additional constants for branching connections mentioned in Section 1 ($\wedge$, $\perp$, $\vee$ and $\top$) is explained.

It is convenient to use, in addition to the operations and constants of BNA, the extensions $\uparrow^m$, I_m and ${}^m\mathsf{X}^n$ of the feedback operation and the identity and transposition constants. These extensions are defined by the axioms R5–R6, B6 and B8–B9, respectively, of BNA (see Section 4.1, Table 1). They are called the block extensions of the feedback operation and these constants. The block extensions of additional constants for branching connections can be defined in the same vein.

In Figure 1, the meaning of the operations and constants of BNA (including the block extensions) is illustrated by means of a graphical representation of networks. We write $f : k \to l$ to indicate that network f has k input ports and l output ports; $k \to l$ is called the sort of f. The input ports are numbered $1, \ldots, k$ and the output ports $1, \ldots, l$. In the graphical representation, they are considered to be numbered from left to right. The networks are drawn with the flow moving from top to bottom. Note that the symbols for the feedback operation and the constants fit with this graphical representation. In Figure 2, the meaning of (block extensions of) the additional constants for branching connections mentioned in Section 1 ($\wedge$, $\perp$, $\vee$ and $\top$) is illustrated by means of a graphical representation. The symbols for these additional constants fit also with the graphical representation. The graphical representations of $\perp^3$ and $\top_2$ reflect to a certain extent the axioms F3 and F4, respectively, for these constants (see e.g. Section 4.1, Table 2).

The operations and constants illustrated above allow to represent all networks (cf. [42]). For example,

$$r_{k,l} = ((\circ_{i=1}^{k-1}(\mathsf{I}_{k-i} \mathbin{+\!\!+} \mathbin{+\!\!+}^i f \mathbin{+\!\!+} \mathsf{I}_{l-i}) \circ \circ_{i=0}^{l-k}(\mathsf{I}_i \mathbin{+\!\!+} \mathbin{+\!\!+}^k f \mathbin{+\!\!+} \mathsf{I}_{l-k-i}) \circ \circ_{i=k-1}^{1}(\mathsf{I}_{l-i} \mathbin{+\!\!+} \mathbin{+\!\!+}^i f \mathbin{+\!\!+} \mathsf{I}_{k-i})) \circ {}^l\mathsf{X}^k) \uparrow^l$$

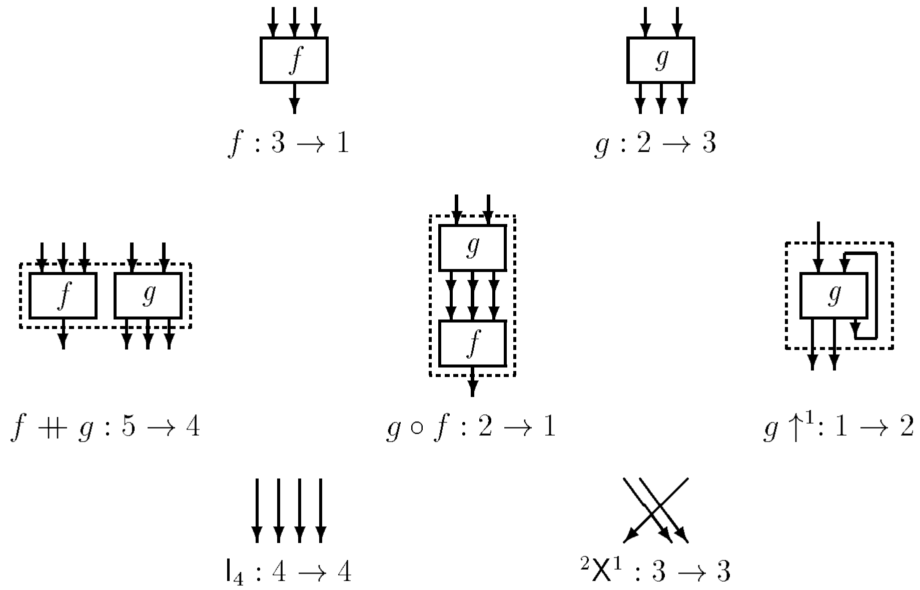


Figure 1: Operations and constants of BNA

Figure 2: Additional constants for branching connections

where $k < l$ and $f : 2 \to 2$, represent a regular network (some abbreviations are used here: iterated sequential composition $\circ_{i=m}^{n} f_i = f_m \circ \ldots \circ f_n$ and parallel composition to the nth $\mathbin{+\!\!+}^n f = f \mathbin{+\!\!+} \ldots \mathbin{+\!\!+} f$ (n times)). The instance $r_{3,4}$ is illustrated in Figure 3.

The graphical illustration of the meaning of the operations and constants of BNA in Figure 1 gives intuitive grounds for the soundness of the axioms of BNA (see Section 4.1, Table 1) for the intended network model. Similarly, the illustration of the meaning of the additional constants for branching connections in Figure 2 makes most additional axioms for these constants (see e.g. Section 4.1, Table 2) plausible.

2.2 Dataflow networks

In the case of dataflow networks, the components are also called cells. The identity connections are called wires and the transposition connections are viewed as crossing wires. The cells are interpreted as processes that consume data at their input ports, compute new data, deliver the new data at their output ports, and then start over again. The wires are interpreted as queues of some kind. The classical kinds considered are firstly queues that deliver data with a neglectible delay and never contain more than one datum, and secondly unbounded, delaying queues. In this paper, they are called *minimal stream delayers* and *stream delayers*, respectively. A stream is a sequence of data consumed or produced by a component of a dataflow network. A

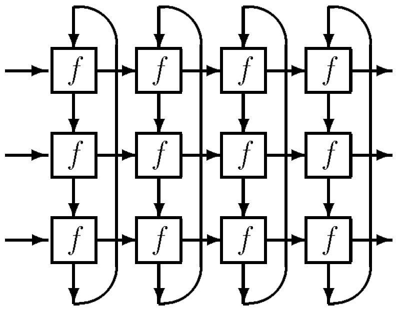


Figure 3: A regular network

flow (of data) is a transformation of a tuple of streams into a tuple of streams. A wire behaves as an identity flow. If the wire is a stream delayer, data pass through it with a time delay. If the wire is a minimal stream delayer, data enter and leave it with a neglectible delay – i.e. within the same time slice in case time is divided into time slices with the length of the time unit used.

In synchronous dataflow networks, the wires are minimal stream delayers. Basic to synchronous dataflow is that there is a global clock. On ticks of the clock, cells can start up the consumption of exactly one datum from each of their input ports and the production of exactly one datum at each of their output ports. A cell that started up with that completes the production of data before the next tick, and it completes the consumption of data as soon as a new datum has been delivered at all input ports. On the first tick following the completion of both, the cell concerned starts up again. In order to start the synchronous dataflow network, every cell has, for each of its output ports, an initial datum available to deliver on the initial tick. The underlying idea of synchronous dataflow is that computation takes a good deal of time, whereas storage and transport of data takes a neglectible deal of time. Phrased differently, data always pass through a wire between two consecutive ticks of the global clock. So minimal stream delayers fit in exactly with this kind of dataflow networks. The semantics of synchronous dataflow networks turns out to be rather simple and unproblematic.

In asynchronous dataflow networks, the wires are stream delayers. The underlying idea of asynchronous dataflow is that computation as well as storage and transport of data takes a good deal of time, which is sometimes more realistic for large systems. In such cases, it is favourable to have computation driven by the arrival of the data needed – instead of by clock ticks. Therefore, there is no global clock in an asynchronous dataflow network. Cells may independently consume data from their input ports, compute new data, and deliver the new data at their output ports. Because it means that there may be data produced by cells but not yet consumed by other cells, this needs wires that are able to buffer an arbitrary amount of data. So stream delayers fit in exactly with this kind of dataflow networks. However, the semantics of asynchronous dataflow networks turns out to be rather problematic. The main semantic problem is a time anomaly, known as the Brock-Ackermann anomaly. With feedback, timing differences in producing data may become important and the time anomaly actually shows that delaying queues do not perfectly fit in with that. Besides, the unbounded queues needed to keep an arbitrary amount of data are unrealistic.

Note that a synchronous dataflow network can be viewed as a extreme case of an asynchronous one, where the queues never contain more than one datum.

Dataflow networks also need branching connections. Their branching structure is more complex than the branching structure of flowcharts. In case of flowcharts, there is a flow of control which is always at one point in the flowchart concerned. In consequence, the interpretation of the branching connections is rather obvious. However, in case of dataflow networks, there is a flow of data which is everywhere in the network. Hence, the interpretation of the branching connections is not immediately clear. In this paper, two kinds of interpretation are considered. For the binary branching connections, they are the *copy/equality test* interpretation and the *split/merge* interpretation. The first kind of interpretation fits in with the idea of permanent flows of data which naturally go in all directions at branchings. Synchronous dataflow reflects this idea most closely. The second kind of interpretation fits in with the idea of intermittent flows of data which go in one direction at branchings. Asynchronous dataflow reflects this idea better. In order to distinguish between the branching constants with these different interpretations, different symbols for $\wedge^m$ and $\vee_m$ are used: $\overset{\circ}{\wedge}{}^m$ and $\underset{\circ}{\vee}{}_m$ for the copy/equality test interpretation, $\overset{\bullet}{\wedge}{}^m$ and $\underset{\bullet}{\vee}{}_m$ for the split/merge interpretation. Likewise, different symbols for the nullary counterparts $\perp^m$ and $\top_m$ are used: $\underset{\circ}{\perp}{}^m$ and $\overset{\circ}{\top}{}_m$ versus $\underset{\bullet}{\perp}{}^m$ and $\overset{\bullet}{\top}{}_m$. $\underset{\circ}{\perp}{}^m$ and $\underset{\bullet}{\perp}{}^m$ are called *sink* and *dummy sink*, respectively; and $\overset{\circ}{\top}{}_m$ and $\overset{\bullet}{\top}{}_m$ are called *source* and *dummy source*, respectively.

In the synchronous case, with minimal stream delayers as identity connections and the copy/equality test interpretation of the branching connections, it turns out that two axioms from Table 2 (A3 and F5) are not valid. Fortunately the others together with two new axioms A3° and F5° (see Section 5.1, Table 3) give a complete set of axioms. The asynchronous case is somewhat problematic owing to the time anomaly that occurs in the model outlined above. Several other models for asynchronous dataflow have been proposed as alternatives, but the valid axioms differ from one model to another. In the process algebra model presented in this paper, with stream delayers as identity connections and the split/merge interpretation of the branching connections, it turns out that five axioms from Table 2 (A5, A7, A10, A11 and F5) are not valid. The question whether there is a complete set of axioms for this model is left open. The relationship between the branching constants from both kinds remains to be investigated as well: the copy/equality test interpretation of the branching connections seems also meaningful in the asynchronous case.

Dataflow networks have been extensively studied, see e.g. [10, 17, 18, 19, 33, 34, 37, 40].

3 Process algebra preliminaries

This section gives a brief summary of the ingredients of process algebra which make up the basis for the process algebra models presented in Sections 4, 5 and 6. We will suppose that the reader is familiar with them. Appropriate references to the literature are included.

We will make use of ACP, introduced in [12], extended with the silent step τ and the abstraction operator τ_I for abstraction. Semantically, we adopt the approach to abstraction, originally proposed for ACP in [29], which is based on branching bisimulation. ACP with this kind of abstraction is called ACP^τ. In ACP with abstraction, processes can be composed by sequential composition, written $P \cdot Q$, alternative composition, written $P + Q$, parallel composition, written $P \parallel Q$, encapsulation, written $\partial_H(P)$, and abstraction, written $\tau_I(P)$. We will also use the following abbreviation. Let $(P_i)_{i \in I}$ be a indexed set of process expressions where $I = \{i_1, \ldots, i_n\}$. Then, we write $\sum_{i \in I} P_i$ for $P_{i_1} + \ldots + P_{i_n}$ if $n > 0$ and δ if $n = 0$. For a systematic introduction to ACP, the reader is referred to [9].

Further we will use the following extensions:

renaming We need the possibility of renaming actions. We will use the renaming operator ρ_f, added to ACP in [1]. Here f is a function that renames actions into actions, δ or τ. The expression $\rho_f(P)$ denotes the process P with every occurrence of an action a replaced by $f(a)$. So the most crucial equation from the defining equations of the renaming operator is $\rho_f(a) = f(a)$.

conditionals We will use the two-armed conditional operator $\lhd \rhd$ as in [3]. The expression $P \lhd b \rhd Q$, is to be read as if b then P else Q. The defining equations are $P \lhd \mathsf{t} \rhd Q = P$ and $P \lhd \mathsf{f} \rhd Q = Q$. Besides, we will use the one-armed conditional operator $:\rightarrow$ as in [3]. The expression $b :\rightarrow P$ can only be performed if $b \neq \mathsf{f}$; it is often referred to as a guarded command. The one-armed conditional operator is defined by $b :\rightarrow P = P \lhd b \rhd \delta$.

iteration We will also use the binary version of Kleene's star operator *, added to ACP in [11], with the defining equation $P^* Q = P \cdot (P^* Q) + Q$. The behaviour of $P^* Q$ is zero or more repetitions of P followed by Q.

early input and process prefixing We will additionally use early input action prefixing and the extension of this binding construct to process prefixing, both added to ACP in [4]. Early input action prefixing is defined by the equation $er_i(x) \ ; P = \sum_{d \in D} r_i(d) \cdot P[d/x]$. We use the extension to processes mainly to express parallel input: $(er_1(x_1) \parallel \ldots \parallel er_n(x_n)) \ ; P$. We have:

$$
\begin{aligned}
(er_1(x_1) \parallel er_2(x_2)) \ ; P \quad &= \sum_{d_1 \in D} r_1(d_1) \cdot (er_2(x_2) \ ; P[d_1/x_1]) \\
&\quad + \sum_{d_2 \in D} r_2(d_2) \cdot (er_1(x_1) \ ; P[d_2/x_2]) \\
(er_1(x_1) \parallel er_2(x_2) \parallel er_3(x_3)) \ ; P \quad &= \sum_{d_1 \in D} r_1(d_1) \cdot ((er_2(x_2) \parallel er_3(x_3)) \ ; P[d_1/x_1]) \\
&\quad + \sum_{d_2 \in D} r_2(d_2) \cdot ((er_1(x_1) \parallel er_3(x_3)) \ ; P[d_2/x_2]) \\
&\quad + \sum_{d_3 \in D} r_3(d_3) \cdot ((er_1(x_1) \parallel er_2(x_2)) \ ; P[d_3/x_3])
\end{aligned}
$$

etc.

communication free merge We will not only use the merge operator ($\parallel$) of ACP, but also the communication free merge operator ($|||$). The communication free merge operator can be viewed as a special instance of the synchronisation merge operator $\parallel_H$ of CSP, also added to ACP in [4], viz. the instance for $H = \emptyset$. It is defined by $P \ ||| \ Q = P \ |||\!\!\_ \ Q + Q \ |||\!\!\_ \ P$, where $|||\!\!\_$ is defined as $\lfloor\!\lfloor$ except that $a \cdot P \ |||\!\!\_ \ Q = a \cdot (P \ ||| \ Q)$. Communication free merge can also be expressed in terms of parallel composition, encapsulation and renaming.

discrete time We need a discrete time extension of ACP with relative timing. We will use the extension introduced in [5], called $\mathrm{ACP_{drt}}$, with abstraction as added to it in [6]. Here we give a brief summary. We refer to [5] and [6] for further details on $\mathrm{ACP_{drt}}$ and $\mathrm{ACP^{\tau}_{drt}}$, respectively.

Time is divided into slices indexed by natural numbers. These time slices represent time intervals of a length which corresponds to the time unit used. We will use the constants a, $\underline{\underline{a}}$ (for each a in some given set of actions), $\underline{\underline{\tau}}$ and $\underline{\underline{\delta}}$, as well as the delay operator σ_{rel}. The process a is a performed in any time slice and $\underline{\underline{a}}$ is a performed in the current time slice. Similarly, $\underline{\underline{\tau}}$ is a silent step performed in the current time slice and $\underline{\underline{\delta}}$ is a deadlock in the current time slice. The process $\sigma_{\mathsf{rel}}(P)$ is P delayed one time slice. In this paper, we use the notations from [2]. In [5], the notations $\mathsf{ats}(a)$, $\mathsf{cts}(a)$ and $\mathsf{cts}(\delta)$ are used instead of a, $\underline{\underline{a}}$ and $\underline{\underline{\delta}}$, respectively. Likewise, in [6], the notation $\mathsf{cts}(\tau)$ is used instead of $\underline{\underline{\tau}}$. The process a is defined in terms $\underline{\underline{a}}$ and σ_{rel} by the equation $a = \underline{\underline{a}} + \sigma_{\mathsf{rel}}(a)$. In a parallel composition $P_1 \parallel \ldots \parallel P_n$ the transition to the next time slice is a simultaneous transition of each of the P_is. For example, $\underline{\underline{\delta}} \parallel \sigma_{\mathsf{rel}}(\underline{\underline{b}})$ will never perform $\underline{\underline{b}}$ because $\underline{\underline{\delta}}$ can neither be delayed nor performed, so $\underline{\underline{\delta}} \parallel \sigma_{\mathsf{rel}}(\underline{\underline{b}}) = \underline{\underline{\delta}}$. However, $\underline{\underline{a}} \parallel \sigma_{\mathsf{rel}}(\underline{\underline{b}}) = \underline{\underline{a}} \cdot \sigma_{\mathsf{rel}}(\underline{\underline{b}})$.

We will also use the above-mentioned extensions of ACP in the setting of $\mathrm{ACP_{drt}}$. The integration of renaming, conditionals, iteration and communication free merge in the discrete time setting is obvious. The integration of early input and process prefixing may seem less clear at first sight, but the relevant equations are simply $\underline{\underline{er}}_i(x) \mathbin{;} P = \sum_{d \in D} \underline{\underline{r}}_i(d) \cdot P[d/x]$ and $\sigma_{\mathsf{rel}}(P) \mathbin{;} Q = \sigma_{\mathsf{rel}}(P \mathbin{;} Q)$.

4 Basic network algebra

BNA is essentially the part of the algebra of flownomials [23] that is common to various classes of networks. In particular, it is common to flowcharts and dataflow networks. The additional constants, needed for branching connections, differ however from one class to another. In this section, BNA is presented. First of all, the signature and axioms of BNA are given. The extension of BNA to the algebra of flownomials is also addressed here. In addition, two models of BNA are described: a data transformer model and a process algebra model. In subsequent sections, extensions of BNA for synchronous and asynchronous dataflow networks are provided.

4.1 Signature and axioms of BNA

Signature

In network algebra, networks are built from other networks – starting with atomic components and a variety of connections. Every network f has a sort $k \to l$, where $k, l \in \mathbb{N}$, associated with it. To indicate this, we use the notation $f : k \to l$. The intended meaning of the sort $k \to l$ is the set of networks with k input ports and l output ports. So $f : k \to l$ expresses that f has k input ports and l output ports.

The sorts of the networks to which an operation of network algebra is applied determine the sort of the resulting network. In addition, there are restrictions on the sorts of the networks to which an operation can be applied. For example, sequential composition can not be applied to two networks of arbitrary sorts because the number of output ports of one should agree with the number of input ports of the other.

The signature of BNA is as follows:

Name	Symbol	Arity
Operations:		
parallel composition	$\mathbin{+\!\!\!+}$	$(k \to l) \times (m \to n) \to (k+m \to l+n)$
sequential composition	$\circ$	$(k \to l) \times (l \to m) \to (k \to m)$
feedback	$\uparrow$	$(m+1 \to n+1) \to (m \to n)$
Constants:		
identity	I	$1 \to 1$
transposition	X	$2 \to 2$

Here k, l, m, n range over $\mathbb{N}$. This means, for example, that there is an instance of the sequential composition operator for each $k, l, m \in \mathbb{N}$.

As mentioned in Section 2, we will also use the block extensions of feedback, identity and transposition. The arity of these auxiliary operations and constants is as follows:

Symbol	Arity
$\uparrow^l$	$(m+l \to n+l) \to (m \to n)$
I_m	$m \to m$
${}^m\mathsf{X}^n$	$m+n \to n+m$

Axioms

The axioms of BNA are given in Table 1. The axioms B1–B6 for $\mathbin{+\!\!\!+}$, $\circ$ and I_m define

B1	$f \mathbin{+\!\!\!+} (g \mathbin{+\!\!\!+} h) = (f \mathbin{+\!\!\!+} g) \mathbin{+\!\!\!+} h$	R1	$g \circ (f \uparrow^m) = ((g \mathbin{+\!\!\!+} \mathsf{I}_m) \circ f) \uparrow^m$
B2	$\mathsf{I}_0 \mathbin{+\!\!\!+} f = f = f \mathbin{+\!\!\!+} \mathsf{I}_0$	R2	$(f \uparrow^m) \circ g = (f \circ (g \mathbin{+\!\!\!+} \mathsf{I}_m)) \uparrow^m$
B3	$f \circ (g \circ h) = (f \circ g) \circ h$	R3	$f \mathbin{+\!\!\!+} (g \uparrow^m) = (f \mathbin{+\!\!\!+} g) \uparrow^m$
B4	$\mathsf{I}_k \circ f = f = f \circ \mathsf{I}_l$	R4	$(f \circ (\mathsf{I}_l \mathbin{+\!\!\!+} g)) \uparrow^m = ((\mathsf{I}_k \mathbin{+\!\!\!+} g) \circ f) \uparrow^n$
B5	$(f \mathbin{+\!\!\!+} f') \circ (g \mathbin{+\!\!\!+} g') = (f \circ g) \mathbin{+\!\!\!+} (f' \circ g')$		for $f : k+m \to l+n,\ g : n \to m$
B6	$\mathsf{I}_k \mathbin{+\!\!\!+} \mathsf{I}_l = \mathsf{I}_{k+l}$	R5	$f \uparrow^0 = f$
B7	${}^k\mathsf{X}^l \circ {}^l\mathsf{X}^k = \mathsf{I}_{k+l}$	R6	$(f \uparrow^l) \uparrow^k = f \uparrow^{k+l}$
B8	${}^k\mathsf{X}^0 = \mathsf{I}_k$		
B9	${}^k\mathsf{X}^{l+m} = ({}^k\mathsf{X}^l \mathbin{+\!\!\!+} \mathsf{I}_m) \circ (\mathsf{I}_l \mathbin{+\!\!\!+} {}^k\mathsf{X}^m)$		
B10	$(f \mathbin{+\!\!\!+} g) \circ {}^m\mathsf{X}^n = {}^k\mathsf{X}^l \circ (g \mathbin{+\!\!\!+} f)$	F1	$\mathsf{I}_k \uparrow^k = \mathsf{I}_0$
	for $f : k \to m,\ g : l \to n$	F2	${}^k\mathsf{X}^k \uparrow^k = \mathsf{I}_k$

Table 1: Axioms of BNA

a strict monoidal category; and together with the additional axioms B7–B10 for ${}^m\mathsf{X}^n$, they define a *symmetric* strict monoidal category (ssmc for short). The remaining axioms R1–R6 and F1–F2 characterize $\uparrow^l$. The axioms R5–R6, B6 and B8–B9 can be regarded as the defining equations of the block extensions of $\uparrow$, I and X, respectively.

The axioms of BNA are sound and complete for networks modulo graph isomorphism (cf. [42]). Using the graphical representation of Section 2.1, it is easy to see that the axioms in Table 1 are sound. By means of the axioms of BNA, each expression can be brought into a normal form

$$((\mathsf{I}_m \mathbin{+\!\!+} x_1 \mathbin{+\!\!+} \ldots \mathbin{+\!\!+} x_k) \circ f) \uparrow^{m_1+\ldots+m_k}$$

where the $x_i : m_i \to n_i$ ($i \in [k]$) are the atomic components of the network and $f : m+n_1+\ldots+n_k \to n+m_1+\ldots+m_k$ is a bijective connection. A network is uniquely represented by a normal form expression up to a permutation of $x_1, \ldots, x_k$. The completeness of the axioms of BNA now follows from the fact that these permutations in a normal form expression are deducible from the axioms of BNA as well.

As a first step towards the stream transformer and process algebra models for dataflow networks described in Sections 5 and 6, a data transformer model and a process algebra model of BNA are provided immediately after the connection with the algebra of flownomials has been addressed.

Extension to the algebra of flownomials

The algebra of flownomials is essentially[1] a conservative extension of BNA. Recall that the algebra of flownomials was not developed for dataflow networks, but for flowcharts. The signature of the algebra of flownomials is obtained by extending the signature of BNA as follows with additional constants for branching connections:

Name	Symbol	Arity	Instances
Additional constants:			
ramification	$\wedge_k$	$1 \to k$	$\begin{cases} \wedge := \wedge_2 \\ \perp := \wedge_0 \end{cases}$
identification	$\vee^k$	$k \to 1$	$\begin{cases} \vee := \vee^2 \\ \top := \vee^0 \end{cases}$

We will restrict our attention to the instances for $k = 0$ and $k = 2$, i.e. $\wedge$, $\perp$, $\vee$ and $\top$. The other instances can be defined in terms of them:

$$\begin{aligned} \wedge_{k+1} &= \wedge \circ (\wedge_k \mathbin{+\!\!+} \mathsf{I}) \\ \vee^{k+1} &= (\vee^k \mathbin{+\!\!+} \mathsf{I}) \circ \vee \end{aligned}$$

It follows from these definitions, together with the axioms A3 and A7 of the algebra of flownomials (see Table 2), that $\wedge_1 = \vee^1 = \mathsf{I}$.

We will use the block extensions of $\wedge$, $\perp$, $\vee$ and $\top$. The arity of these auxiliary constants is as follows:

[1] For naming ports, an arbitrary monoid is used in the algebra of flownomials whereas the monoid of natural numbers is used in BNA.

Symbol	Arity
$\wedge^m$	$m \to 2m$
$\perp^m$	$m \to 0$
$\vee_m$	$2m \to m$
$\top_m$	$0 \to m$

The axioms for the additional constants of the algebra of flownomials are given in Table 2. These axioms where chosen in order to describe the branching structure of

A1	$(\vee_m \mathbin{+\!\!+} \mathsf{I}_m) \circ \vee_m = (\mathsf{I}_m \mathbin{+\!\!+} \vee_m) \circ \vee_m$	A5	$\wedge^m \circ (\wedge^m \mathbin{+\!\!+} \mathsf{I}_m) = \wedge^m \circ (\mathsf{I}_m \mathbin{+\!\!+} \wedge^m)$
A2	${}^m\mathsf{X}^m \circ \vee_m = \vee_m$	A6	$\wedge^m \circ {}^m\mathsf{X}^m = \wedge^m$
A3	$(\top_m \mathbin{+\!\!+} \mathsf{I}_m) \circ \vee_m = \mathsf{I}_m$	A7	$\wedge^m \circ (\perp^m \mathbin{+\!\!+} \mathsf{I}_m) = \mathsf{I}_m$
A4	$\vee_m \circ \perp^m = \perp^m \mathbin{+\!\!+} \perp^m$	A8	$\top_m \circ \wedge^m = \top_m \mathbin{+\!\!+} \top_m$
A9	$\top_m \circ \perp^m = \mathsf{I}_0$		
A10	$\vee_m \circ \wedge^m = (\wedge^m \mathbin{+\!\!+} \wedge^m) \circ (\mathsf{I}_m \mathbin{+\!\!+} {}^m\mathsf{X}^m \mathbin{+\!\!+} \mathsf{I}_m) \circ (\vee_m \mathbin{+\!\!+} \vee_m)$		
A11	$\wedge^m \circ \vee_m = \mathsf{I}_m$		
A12	$\top_0 = \mathsf{I}_0$	A16	$\perp^0 = \mathsf{I}_0$
A13	$\top_{m+n} = \top_m \mathbin{+\!\!+} \top_n$	A17	$\perp^{m+n} = \perp^m \mathbin{+\!\!+} \perp^n$
A14	$\vee_0 = \mathsf{I}_0$	A18	$\wedge^0 = \mathsf{I}_0$
A15	$\vee_{m+n} = (\mathsf{I}_m \mathbin{+\!\!+} {}^n\mathsf{X}^m \mathbin{+\!\!+} \mathsf{I}_n) \circ (\vee_m \mathbin{+\!\!+} \vee_n)$	A19	$\wedge^{m+n} = (\wedge^m \mathbin{+\!\!+} \wedge^n) \circ (\mathsf{I}_m \mathbin{+\!\!+} {}^m\mathsf{X}^n \mathbin{+\!\!+} \mathsf{I}_n)$
F3	$\vee_m \uparrow^m = \perp^m$	F4	$\wedge^m \uparrow^m = \top_m$
F5	$((\mathsf{I}_m \mathbin{+\!\!+} \wedge^m) \circ ({}^m\mathsf{X}^m \mathbin{+\!\!+} \mathsf{I}_m) \circ (\mathsf{I}_m \mathbin{+\!\!+} \vee_m)) \uparrow^m = \mathsf{I}_m$		

Table 2: Additional axioms for flowcharts

flowcharts. The axioms A12–A19 can be regarded as the defining equations of the block extentions of $\wedge$, $\perp$, $\vee$ and $\top$.

The standard model for the interpretation of flowcharts is the model $\mathbb{R}\mathsf{el}(D)$ of relations over a set D (cf. [23, 41]). All axioms of the algebra of flownomials (Tables 1 and 2) hold in this model. The algebraic structure defined by the axioms of BNA (Table 1) was introduced in [42] under the name of *biflow*. In [43] it is called *$a\alpha$-ssmc with feedback*. The algebraic structure defined by the axioms of the algebra of flownomials (Tables 1 and 2) is called *$d\delta$-ssmc with feedback* in [43].

4.2 Data transformer model of BNA

In this subsection, a data transformer model is described. A parallel data transformer $f : m \to n$ acts on an m-tuple of input data and produces an n-tuple of output data. Parallel composition, sequential composition and feedback operators as well as identity and transposition constants are defined on parallel data transformers. All axioms of BNA (Table 1) hold in the resulting model.

Definition 4.1 (data transformer model of BNA)
A *parallel data transforming relation* $f \in \mathsf{Rel}(S)(m,n)$ is a relation

$$f \subseteq S^m \times S^n$$

where S is a set of data. $\mathsf{Rel}(S)$ denotes the indexed family of data transforming relations $(\mathsf{Rel}(S)(m,n))_{\mathbb{N}\times\mathbb{N}}$.

The operations and constants of BNA are defined on $\mathsf{Rel}(S)$ as follows:

Name	Notation		
parallel composition	$f \mathbin{+\!\!+} g$	$\in \mathsf{Rel}(S)(m+p,n+q)$	for $f\in\mathsf{Rel}(S)(m,n)$, $g\in\mathsf{Rel}(S)(p,q)$
sequential composition	$f\circ g$	$\in \mathsf{Rel}(S)(m,p)$	for $f\in\mathsf{Rel}(S)(m,n)$, $g\in\mathsf{Rel}(S)(n,p)$
feedback	$f\uparrow^p$	$\in \mathsf{Rel}(S)(m,n)$	for $f\in\mathsf{Rel}(S)(m+p,n+p)$
identity	I_n	$\in \mathsf{Rel}(S)(n,n)$	
transposition	${}^m\mathsf{X}^n$	$\in \mathsf{Rel}(S)(m+n,n+m)$	

Definition

$$
\begin{aligned}
f \mathbin{+\!\!+} g &= \{\langle x^\frown y, z^\frown w\rangle \mid x\in S^m, y\in S^p, z\in S^n, w\in S^q, \langle x,z\rangle\in f\wedge\langle y,w\rangle\in g\}\\
f\circ g &= \{\langle x,y\rangle \mid x\in S^m, y\in S^p, \exists z\in S^n\cdot\langle x,z\rangle\in f\wedge\langle z,y\rangle\in g\}\\
f\uparrow^p &= \{\langle x,y\rangle \mid x\in S^m, y\in S^n, \exists z\in S^p\cdot\langle x^\frown z, y^\frown z\rangle\in f\}\\
\mathsf{I}_n &= \{\langle x,x\rangle \mid x\in S^n\}\\
{}^m\mathsf{X}^n &= \{\langle x^\frown y, y^\frown x\rangle \mid x\in S^m, y\in S^n\}
\end{aligned}
$$

□

These definitions are very straightforward. Note that this data transformer model has a global crash property: if a component of a network fails to produce output, the whole network fails to produce output.

Theorem 4.2 $(\mathsf{Rel}(S), \mathbin{+\!\!+}, \circ, \uparrow, \mathsf{I}, \mathsf{X})$ *is a model of BNA.*

Proof: The proof is a matter of straightforward calculation using only elementary set theory. □

Additional branching constants can be defined such that the resulting expanded model satisfies most axioms of the algebra of flownomials (Tables 1 and 2). One such set of branching constants is closely related to the one that is used in the design of (nondeterministic) SCAs [44]. The corresponding expanded model is principally the stream transformer model for synchronous dataflow networks described in Section 5 where an abstraction is made from the internals of the transformers: arbitrary data is transformed instead of streams of data. However, $\top_m$ must be interpreted like ♀_m in this data transformer model, to keep up relationships with SCAs, whereas it is interpreted as ⚲_m in the stream transformer model for synchronous dataflow networks.

4.3 Process algebra model of BNA

Network algebra can be regarded as being built on top of process algebra.

Let D be a fixed, but arbitrary, set of data. D is a parameter of the model. The processes use the standard actions $r_i(d)$, $s_i(d)$ and $c_i(d)$ for $d \in D$ only. They stand for read, send and communicate, respectively, datum d at port i. On these actions, communication is defined such that $r_i(d) \mid s_i(d) = c_i(d)$ (for all $i \in \mathbb{N}$ and $d \in D$). In all other cases, it yields δ.

We write $H(i)$, where $i \in \mathbb{N}$, for the set $\{r_i(d) \mid d \in D\} \cup \{s_i(d) \mid d \in D\}$ and $I(i)$ for $\{c_i(d) \mid d \in D\}$. In addition, we write $H(i,j)$ for $H(i) \cup H(j)$, $H(i+[k])$ for $H(i+1) \cup \ldots \cup H(i+k)$ and $H(i+[k], j+[l])$ for $H(i+[k]) \cup H(j+[l])$. The abbreviations $I(i,j)$, $I(i+[k])$ and $I(i+[k], j+[l])$ are used analogously.

$in(i/j)$ denotes the renaming function defined by

$$\begin{array}{lcll} in(i/j)(r_i(d)) & = & r_j(d) & \text{for } d \in D \\ in(i/j)(a) & = & a & \text{for } a \notin \{r_i(d) \mid d \in D\} \end{array}$$

So $in(i/j)$ renames port i into j in read actions. $out(i/j)$ is defined analogously, but renames send actions. We write $in(i+[k]/j+[k])$ for $in(i+1/j+1) \circ \ldots \circ in(i+k/j+k)$ and $in([k]/j+[k])$ for $in(0+[k]/j+[k])$. The abbreviations $out(i+[k]/j+[k])$ and $out([k]/j+[k])$ are used analogously.

Definition 4.3 (process algebra model of BNA)
A *network* $f \in \mathsf{Proc}(D)(m,n)$ is a triple

$$f = (m, n, P)$$

where P is a process with actions in $\{r_i(d) \mid i \in [m], d \in D\} \cup \{s_i(d) \mid i \in [n], d \in D\}$. $\mathsf{Proc}(D)$ denotes the indexed family of sets $(\mathsf{Proc}(D)(m,n))_{\mathbb{N} \times \mathbb{N}}$.

A *wire* is a network $\mathsf{I} = (1, 1, w_1^1)$, where w_1^1 satisfies:

for all networks $f = (m, n, P)$ and $u, v > \max(m, n)$,

(P1) $\tau_{I(u,v)}(\partial_{H(v,u)}(w_v^u \parallel w_u^v)) \mathbin{|||} P = P$

(P2) $\tau_{I(u,v)}(\partial_{H(u,v)}((\rho_{in(i/u)}(P) \mathbin{|||} w_v^i) \parallel w_u^v)) = P \quad$ for all $i \in [m]$

(P3) $\tau_{I(u,v)}(\partial_{H(u,v)}((\rho_{out(j/v)}(P) \mathbin{|||} w_j^u) \parallel w_u^v)) = P \quad$ for all $j \in [n]$

where $w_v^u = \rho_{in(1/u)}(\rho_{out(1/v)}(w_1^1))$

The operations and constants of BNA are defined on $\mathsf{Proc}(D)$ as follows:

Name	Notation		
parallel composition	$f \mathbin{+\!\!\!+} g$	$\in \mathsf{Proc}(D)(m+p, n+q)$	for $f \in \mathsf{Proc}(D)(m,n)$, $g \in \mathsf{Proc}(D)(p,q)$
sequential composition	$f \circ g$	$\in \mathsf{Proc}(D)(m,p)$	for $f \in \mathsf{Proc}(D)(m,n)$, $g \in \mathsf{Proc}(D)(n,p)$
feedback	$f \uparrow^p$	$\in \mathsf{Proc}(D)(m,n)$	for $f \in \mathsf{Proc}(D)(m+p, n+p)$
identity	I_n	$\in \mathsf{Proc}(D)(n,n)$	
transposition	${}^m\mathsf{X}^n$	$\in \mathsf{Proc}(D)(m+n, n+m)$	

Definition

$$(m,n,P) \mathbin{+\!\!+} (p,q,Q) = (m+p,n+q,R) \quad \text{where } R = P \mathbin{|||} \rho_{in([p]/m+[p])}(\rho_{out([q]/n+[q])}(Q))$$

$$(m,n,P) \circ (n,p,Q) = (m,p,R) \quad \text{where, for } u = \max(m,p),\ v = u+n,$$
$$R = \tau_{I(u+[n],v+[n])}(\partial_{H(u+[n],v+[n])}((\rho_{out([n]/u+[n])}(P) \mathbin{|||} \rho_{in([n]/v+[n])}(Q)) \parallel w_{v+1}^{u+1} \parallel \ldots \parallel w_{v+n}^{u+n}))$$

$$(m+p,n+p,P)\uparrow^p = (m,n,Q) \quad \text{where, for } u = \max(m,n),\ v = u+p,$$
$$Q = \tau_{I(u+[p],v+[p])}(\partial_{H(u+[p],v+[p])}(\rho_{in(m+[p]/v+[p])}(\rho_{out(n+[p]/u+[p])}(P)) \parallel w_{v+1}^{u+1} \parallel \ldots \parallel w_{v+p}^{u+p}))$$

$$\mathsf{I}_n = (n,n,P) \quad \text{where } P = \begin{cases} w_1^1 \mathbin{|||} \ldots \mathbin{|||} w_n^n & \text{if } n > 0 \\ \tau_{I(1,2)}(\partial_{H(1,2)}(w_2^1 \parallel w_1^2)) & \text{otherwise} \end{cases}$$

$${}^m\mathsf{X}^n = (m+n,n+m,P) \quad \text{where } P = \begin{cases} w_{n+1}^1 \mathbin{|||} \ldots \mathbin{|||} w_{n+m}^m \mathbin{|||} w_1^{m+1} \mathbin{|||} \ldots \mathbin{|||} w_n^{m+n} & \text{if } m+n > 0 \\ \tau_{I(1,2)}(\partial_{H(1,2)}(w_2^1 \parallel w_1^2)) & \text{otherwise} \end{cases}$$

□

The conditions (P1)–(P3) are rather obscure at first sight, but see the remark at the end of this section. The definitions of sequential composition and feedback illustrate clearly the differences between the mechanisms for using ports in network algebra and process algebra. In network algebra the ports that become internal after composition are hidden. In process algebra based models these ports are still visible; a special operator must be used to hide them. For typical wires, $\tau_{I(1,2)}(\partial_{H(1,2)}(w_2^1 \parallel w_1^2))$ equals δ, $\tau \cdot \delta$ or $\underline{\underline{\tau}} \cdot \delta$ (the latter only in case $\mathrm{ACP}^{\tau}_{\mathrm{drt}}$ is used).

In the description of a process algebra model of BNA given above, all constants and operators used are common to ACP^{τ} and $\mathrm{ACP}^{\tau}_{\mathrm{drt}}$ or belong to a few of their mutual (conservative) extensions mentioned in Section 3 (viz. renaming and communication free merge). As a result, we can specialize this general model for a specific kind of networks using either ACP^{τ} or $\mathrm{ACP}^{\tau}_{\mathrm{drt}}$; with further extensions at need. On the other hand, we can obtain general results on these process algebra models: results that only depend on properties that are common to ACP^{τ} and $\mathrm{ACP}^{\tau}_{\mathrm{drt}}$ or properties of the mutual extensions used above.

Theorem 4.4 $(\mathsf{Proc}(D), \mathbin{+\!\!+}, \circ, \uparrow, \mathsf{I}, \mathsf{X})$ *is a model of BNA.*

Proof: According to [43], there is an algebra equivalent to BNA (the algebra of LR-flow over $\mathbb{B}\mathsf{i}$), but having two renumbering operations, for (bijectively) renumbering input ports and output ports, instead of the transposition constant and the sequential composition operation of BNA. Renumbering is just renaming in the corresponding process algebra model. The crucial axioms concerning the constant I_n in the equational theory of that algebra follow immediately from the conditions (P1)–(P3) on wires in Definition 4.3. For quite a few axioms from this equational theory, the proof that they are satisfied by the process algebra model is a matter of simple calculation using only elementary properties of renaming, communication free merge, or parallel composition and renaming. For the remaining axioms, reminiscent of the axioms R1–R4 of BNA, the proof is a matter of straightforward calculation using in addition properties of parallel composition and encapsulation or abstraction. All properties concerned are common to ACP^{τ} and $\mathrm{ACP}^{\tau}_{\mathrm{drt}}$ or properties of the mutual extensions used in Definition 4.3. □

So if we select a specific wire, such as $\mathtt{msd}_1^1$ in Section 5 and $\mathtt{sd}_1^1$ in Section 6, we have obtained a model of BNA if the conditions (P1)–(P3) are satisfied by the wire concerned. It is worth mentioning that the conditions (P1)–(P3) are equivalent to the axioms B2 and B4 of BNA: (P1) corresponds to $\mathsf{I}_0 \mathbin{+\!\!\!+} f = f = f \mathbin{+\!\!\!+} \mathsf{I}_0$, (P2) to $\mathsf{I}_m \circ f = f$, and (P3) to $f = f \circ \mathsf{I}_n$.

5 Synchronous dataflow networks

In this section, an extension of BNA for synchronous dataflow networks is presented. In the first place, the additional constants and axioms for synchronous dataflow are given. After that, the adaptation of the data transformer model of Section 4.2 to synchronous dataflow networks, resulting in a stream transformer model for synchronous dataflow, is described. Finally, the specialization of the process algebra model of Section 4.3 for synchronous dataflow networks is described.

5.1 Additional constants and axioms

The signature of the extension of BNA for synchronous dataflow networks is obtained by extending the signature of BNA as follows with additional constants for branching connections:

Name	Symbol	Arity
Additional constants:		
copy	$\wedge^m$	$m \to 2m$
sink	$\bot^m$	$m \to 0$
equality test	$\vee_m$	$2m \to m$
dummy source	$\top^{\bullet}_m$	$0 \to m$

The symbols $\wedge^m$, $\bot^m$ and $\vee_m$ indicate that the copy/equality test interpretation is intended here. For technical reasons, which are explained at the end of Section 5.2, $\top^{\bullet}_m$ is used instead of $\top_m$.

The axioms for these additional constants are given in Table 3. These axioms agree with those for the additional constants of the algebra of flownomials (Table 2) with two exceptions: A3 and F5 are replaced by A3° and F5°.

In the next two subsections, the models introduced in Section 4 are specialized to describe the semantics of the synchronous dataflow networks.

5.2 Stream transformer model for synchronous dataflow

In this subsection, an adaptation of the data transformer model of BNA (Section 4.2) for synchronous dataflow is given.

In Section 4.2, no assumptions about the nature of the transformers were made. Here the nature of the transformers needed for synchronous dataflow networks is made precise, resulting in the definition of quasiproper stream transformers. The feedback operation is adapted to reflect a special characteristic of feedback in synchronous

A1	$(\vee_m \mathbin{+\!\!+} \mathsf{I}_m) \circ \vee_m = (\mathsf{I}_m \mathbin{+\!\!+} \vee_m) \circ \vee_m$	A5	$\wedge^m \circ (\wedge^m \mathbin{+\!\!+} \mathsf{I}_m) = \wedge^m \circ (\mathsf{I}_m \mathbin{+\!\!+} \wedge^m)$
A2	${}^m\mathsf{X}^m \circ \vee_m = \vee_m$	A6	$\wedge^m \circ {}^m\mathsf{X}^m = \wedge^m$
A3°	$(\top_m \mathbin{+\!\!+} \mathsf{I}_m) \circ \vee_m = \perp^m \circ \top_m$	A7	$\wedge^m \circ (\perp^m \mathbin{+\!\!+} \mathsf{I}_m) = \mathsf{I}_m$
A4	$\vee_m \circ \perp^m = \perp^m \mathbin{+\!\!+} \perp^m$	A8	$\top_m \circ \wedge^m = \top_m \mathbin{+\!\!+} \top_m$
A9	$\top_m \circ \perp^m = \mathsf{I}_0$		
A10	$\vee_m \circ \wedge^m = (\wedge^m \mathbin{+\!\!+} \wedge^m) \circ (\mathsf{I}_m \mathbin{+\!\!+} {}^m\mathsf{X}^m \mathbin{+\!\!+} \mathsf{I}_m) \circ (\vee_m \mathbin{+\!\!+} \vee_m)$		
A11	$\wedge^m \circ \vee_m = \mathsf{I}_m$		
A12	$\top_0 = \mathsf{I}_0$	A16	$\perp^0 = \mathsf{I}_0$
A13	$\top_{m+n} = \top_m \mathbin{+\!\!+} \top_n$	A17	$\perp^{m+n} = \perp^m \mathbin{+\!\!+} \perp^n$
A14	$\vee_0 = \mathsf{I}_0$	A18	$\wedge^0 = \mathsf{I}_0$
A15	$\vee_{m+n} = (\mathsf{I}_m \mathbin{+\!\!+} {}^n\mathsf{X}^m \mathbin{+\!\!+} \mathsf{I}_n) \circ (\vee_m \mathbin{+\!\!+} \vee_n)$	A19	$\wedge^{m+n} = (\wedge^m \mathbin{+\!\!+} \wedge^n) \circ (\mathsf{I}_m \mathbin{+\!\!+} {}^m\mathsf{X}^n \mathbin{+\!\!+} \mathsf{I}_n)$
F3	$\vee_m \uparrow^m = \perp^m$	F4	$\wedge^m \uparrow^m = \top_m$
F5°	$((\mathsf{I}_m \mathbin{+\!\!+} \wedge^m) \circ ({}^m\mathsf{X}^m \mathbin{+\!\!+} \mathsf{I}_m) \circ (\mathsf{I}_m \mathbin{+\!\!+} \vee_m)) \uparrow^m = \perp^m \circ \top_m$		

Table 3: Additional axioms for synchronous dataflow networks

dataflow networks: data in the feedback loop produced in one time slice is not used to produce new data before the next time slice.

The model $\mathsf{Rel}(S)$ of Section 4.2 is a general model. In case of dataflow, streams of data are transformed. This means that

$$S = (D \cup \{\surd\})^\infty = \mathbb{N} \to (D \cup \{\surd\})$$

for some set of data D, $\surd \notin D$. For a stream $x \in S$ and $k \in \mathbb{N}$, $x(k)$ is the datum occurring in that stream on the k-th tick of the global clock if $x(k) \in D$. The absence of a datum is represented by $\surd$; so $x(k) = \surd$ indicates that no datum occurs in stream x on the k-th tick. This may happen, for example, with the equality test $\vee_1$: no datum is delivered on the k-th tick unless equal data are offered at its input ports on that tick. Owing to this approach to deal with the absence of data, it is quite natural in case of synchronous dataflow to look at finite streams as infinite ones where no datum occurs from a certain tick. This point of view has the additional advantage that the relevant definitions can be kept simple. However, it is unnatural to uphold this view-point for asynchronous dataflow.

The stream transformers used to model the cells in synchronous dataflow networks have a "dependency on the past" property which is captured by the following definition.

Definition 5.1 (proper stream transformer)
A stream transformer $f \in \mathsf{Rel}(S)(m,n)$ is *proper* (or *determined by the past*) if

$$\begin{array}{l} \forall x \in S^m \cdot \forall x' \in S^m \cdot \\ \quad \{y(0) \mid y \in S^n, \langle x,y\rangle \in f\} = \{y'(0) \mid y' \in S^n, \langle x',y'\rangle \in f\} \wedge \\ \quad \forall k \in \mathbb{N} \cdot x(0..k) = x'(0..k) \;\Rightarrow \\ \quad\; \{y(0..k+1) \mid y \in S^n, \langle x,y\rangle \in f\} = \{y'(0..k+1) \mid y' \in S^n, \langle x',y'\rangle \in f\} \end{array}$$

□

Note that this property reduces at the beginning to a "constant output initially" property.

The proper stream transformers fail to include constants for connections such as I, X, $\curlywedge$ and $\curlyvee$, because their intended meaning is to let data pass through them with a neglectible delay. Because at least the constants I and X are necessary in order to define a network algebra, stream transformers built from proper stream transformers and stream transformers with input and output ports that are directly connected must be allowed. The resulting stream transformers are called quasiproper stream transformers. A similar notion is used in [10].

Definition 5.2 (direct connection)
Two ports $i \in [m]$ and $j \in [n]$ are *directly connected* via a stream transformer $f \in \mathsf{Rel}(S)(m,n)$ if

$$\begin{array}{l} \forall (x_1, \ldots, x_m) \in S^m \cdot \forall (y_1, \ldots, y_n) \in S^n \cdot \\ \quad \langle (x_1, \ldots, x_m), (y_1, \ldots, y_n) \rangle \in f \;\Rightarrow\; x_i = y_j \end{array}$$

We write $dc(f)$ for the set $\{(i,j) \mid i \text{ is directly connected with } j \text{ via } f\}$

A stream transformer $f \in \mathsf{Rel}(S)(m,n)$ is a *direct connection* if

$$\forall i \in [m] \cdot \exists j \in [n] \cdot (i,j) \in dc(f) \wedge \forall j \in [n] \cdot \exists i \in [m] \cdot (i,j) \in dc(f)$$

□

Definition 5.3 (quasiproper stream transformer)
A stream transformer in $\mathsf{Rel}(S)(m,n)$ is *quasiproper* if it can be described by an expression of the form

$$h \circ (\mathsf{I}_k \mathbin{+\!\!+} \curlywedge^{m-(k+l)} \mathbin{+\!\!+} \mathsf{I}_l) \circ (f \mathbin{+\!\!+} g) \circ (\mathsf{I}_{k'} \mathbin{+\!\!+} \curlyvee_{n-(k'+l')} \mathbin{+\!\!+} \mathsf{I}_{l'}) \circ h'$$

where $f \in \mathsf{Rel}(S)(m-l, n-l')$ is a proper stream transformer, $g \in \mathsf{Rel}(S)(m-k, n-k')$ is a direct connection, and $h \in \mathsf{Rel}(S)(m,m)$ and $h' \in \mathsf{Rel}(S)(n,n)$ are bijective direct connections. The constants $\curlywedge^n \in \mathsf{Rel}(S)(n, n+n)$ and $\curlyvee_n \in \mathsf{Rel}(S)(n+n, n)$ used here are the ones defined below in Definition 5.4. The restriction of $\mathsf{Rel}(S)$ to quasiproper stream transformers is denoted by $\mathsf{QRel}(S)$. The further restriction of $\mathsf{QRel}(S)$ to functions is denoted by $\mathsf{QFn}(S)$. □

With $\mathsf{QFn}(S)$ only deterministic dataflow networks can be modelled, while $\mathsf{QRel}(S)$ covers non-deterministic dataflow as well. If S is a set of streams of data, i.e. $S = (D \cup \{\surd\})^\infty$ for some set of data D, the constants of BNA as defined on $\mathsf{Rel}(S)$ in Section 4.2 are quasiproper functions. So the identity and transposition constants are in $\mathsf{QFn}(S)$ and $\mathsf{QRel}(S)$. In addition, both $\mathsf{QFn}(S)$ and $\mathsf{QRel}(S)$ are closed under the parallel and sequential composition operations as defined on $\mathsf{Rel}(S)$. As mentioned before, the feedback operation as defined on $\mathsf{Rel}(S)$ does not model feedback in synchronous dataflow networks properly. A related problem is that $\mathsf{QFn}(S)$ is not closed under this feedback operation. All this means that only a more appropriate feedback operation and the additional constants for synchronous dataflow have to be defined.

Definition 5.4 (stream transformer model for synchronous dataflow)
The parallel and sequential composition operations on $\mathsf{QRel}(S)$ are the restrictions of the parallel and sequential composition operations on $\mathsf{Rel}(S)$ to $\mathsf{QRel}(S)$. The identity and transposition constants in $\mathsf{QRel}(S)$ are the ones in $\mathsf{Rel}(S)$.

The feedback operation is redefined on $\mathsf{QRel}(S)$ as follows:

Name	Notation	
feedback	$f \uparrow^p \; \in \mathsf{QRel}(S)(m,n)$	for $f \in \mathsf{QRel}(S)(m+p,n+p)$

Definition

$$f \uparrow^1 \;=\; \begin{cases} \{\langle x,y\rangle \mid x \in S^m, y \in S^n, \exists z \in S \cdot \langle x^\frown z, y^\frown z\rangle \in f\} & \text{if } (m+1,n+1) \notin dc(f) \\ (\mathsf{I}_m \mathbin{+\!\!\!+} \top_1) \circ f \circ (\mathsf{I}_n \mathbin{+\!\!\!+} \bot^1) & \text{otherwise} \end{cases}$$

for $p \neq 1$, $\uparrow^p$ is defined by the equations occurring as axioms R5–R6 of BNA.

The constants $\top_n \in \mathsf{QRel}(S)(0,n)$ and $\bot^n \in \mathsf{QRel}(S)(n,0)$ used here are the ones defined right away.

The additional constants for synchronous dataflow are defined on $\mathsf{QRel}(S)$ as follows:

Name	Notation
copy	$\curlywedge^n \; \in \mathsf{QRel}(S)(n,n+n)$
sink	$\bot^n \; \in \mathsf{QRel}(S)(n,0)$
equality test	$\curlyvee_n \; \in \mathsf{QRel}(S)(n+n,n)$
source	$\top_n \; \in \mathsf{QRel}(S)(0,n)$

Definition

$$\curlywedge^n = \{\langle x, x^\frown x\rangle \mid x \in S^n\}$$

$$\bot^n = \{\langle x, ()\rangle \mid x \in S^n\}$$

$$\curlyvee_n = \{\langle (x_1,\ldots,x_n,y_1,\ldots,y_n),(x_1 \,\&\, y_1,\ldots,x_n \,\&\, y_n)\rangle \mid (x_1,\ldots,x_n) \in S^n, (y_1,\ldots,y_n) \in S^n\}$$
where $(x \,\&\, y)(k) = x(k)$ if $x(k) = y(k)$ and $(x \,\&\, y)(k) = \surd$ otherwise

$$\top_n = \{\langle (), (\surd^\infty,\ldots,\surd^\infty)\rangle\}$$

□

In Definition 4.1, the feedback operation was defined such that, for each data transformer f, the feedback loop behaves as the greatest fixpoint of f relative to the input stream of $f\uparrow$. In case of proper stream transformers, there is always a unique fixpoint provided the transformer is a function or a continuous relation (with respect to the prefixes of streams). It means that the feedback loop is also the least fixpoint. This is needed to model feedback in synchronous dataflow networks properly; for otherwise it does not agree with the operational understanding that it is iteratively feeding the network concerned with data produced by it in the previous step. The adaptation of the feedback operation given in Definition 5.4 is needed to get a unique fixpoint in case of quasiproper stream transformers as well. It also guarantees that $\mathsf{QFn}(S)$ is closed under feedback. Because $\wedge\uparrow$ now produces a dummy stream, it equals the dummy source. For this reason, $\bullet$ is used instead of $\circ$ as constant for synchronous dataflow. Note that this stream transformer model does not have the global crash property of the data transformer model from Section 4.2: if a component of a network fails to produce output on some tick of the global clock, the effect is merely that the components connected to the port(s) concerned will fail to produce output on some future tick.

Theorem 5.5 $(\mathsf{QFn}(S), \mathbin{+\!\!\!+}, \circ, \uparrow, \mathsf{I}, \mathsf{X})$ *is a model of BNA. The constants* $\wedge, \bot, \vee, \bullet$ *satisfy the additional axioms for synchronous dataflow networks (Table 3).*

Proof: For the first part, it is enough to prove R1–R4 and F1–F2. According to [21, 22], it suffices to prove R1–R4 for $m = 1$, and R4 additionally for $k = l = 1$ and $g = {}^1\mathsf{X}^1$. The proofs concerned are straightforward proofs by case distinction – the cases depending on whether the ports relevant to the feedback loop are directly connected or not. The second part is a matter of tedious, but simple calculation. □

5.3 Process algebra model for synchronous dataflow

In this subsection, the specialization of the process algebra model of BNA (Section 4.3) for synchronous dataflow networks is given. In this case, we will make use of $\mathrm{ACP}^{\tau}_{\mathrm{drt}}$. Recall that $\mathrm{ACP}^{\tau}_{\mathrm{drt}}$ is $\mathrm{ACP}_{\mathrm{drt}}$ – the discrete relative time extension of ACP – extended with abstraction based on branching bisimulation.

In Section 4.3, only a few assumptions about wires and atomic cells were made. Here it is first explained how these ingredients are actualized for synchronous dataflow networks. Because of the crucial role of the time slices determined by the ticks of a global clock, discrete-time process algebra is used.

Definition 5.6 (wires and atomic cells in synchronous dataflow networks)
In the synchronous case, the identity constant, called the *minimal stream delayer*, is the wire $\mathsf{I}_1 = (1, 1, \mathtt{msd}^1_1)$ where $\mathtt{msd}^1_1$ is defined by

$$\mathtt{msd}^1_1 = \underline{\underline{\tau}} \cdot (er_1(x) \; ; \; \underline{\underline{s}}_1(x)) \cdot \sigma_{\mathsf{rel}}(\mathtt{msd}^1_1)$$

The constants I_n, for $n \neq 1$, and ${}^m\mathsf{X}^n$ are defined by the equations occurring as axioms B6 and B8–B9, respectively, of Table 1.

In the synchronous case, the deterministic cell computing a function $f : D^m \to D^n$, and having $\vec{a} = (a_1, \ldots, a_n) \in D^n$ as its initial output tuple, is the network $C_f(\vec{a}) = (m, n, P_f(\vec{a}))$ where P_f is defined by

$$\begin{aligned} P_f(\vec{a}) &= \underline{\underline{\tau}} \cdot (Out(\vec{a}) \mathbin{|||} ((er_1(x_1) \parallel \ldots \parallel er_m(x_m)) \; ; \; \sigma_{\mathsf{rel}}(P_f(f(x_1, \ldots, x_m))))) \\ \text{where } Out(\vec{a}) &= \underline{\underline{s}}_1(a_1) \parallel \ldots \parallel \underline{\underline{s}}_n(a_n) \end{aligned}$$

The non-deterministic cell computing a (finitely branching) relation $R \subseteq D^m \times D^n$, and having $A \subseteq D^n$ as its set of possible initial output tuples, is the network $C_R(A) = (m, n, P_R(A))$ where P_R is defined by

$$
\begin{aligned}
P_R(A) &= \underline{\underline{\tau}} \cdot (Out(A) \mathbin{|||} ((er_1(x_1) \parallel \ldots \parallel er_m(x_m)) \,;\, \sigma_{\mathsf{rel}}(P_R(R(x_1, \ldots, x_m))))) \\
\text{where } Out(A) &= \underline{\underline{\tau}} \lhd A = \emptyset \rhd \sum_{(a_1,\ldots,a_n)\in A} (\underline{\underline{s}}_1(a_1) \parallel \ldots \parallel \underline{\underline{s}}_1(a_n))
\end{aligned}
$$

The restriction of $\mathsf{Proc}(D)$ to the processes that can be built under this actualization is denoted by $\mathsf{SProc}(D)$. $\Box$

The definition of $\mathtt{msd}_1^1$ expresses the following. The process $\mathtt{msd}_1^1$ waits until a datum is offered at its input port. When a datum is available at the input port, $\mathtt{msd}_1^1$ delivers the datum at its output port in the same time slice. From the next time slice, it proceeds with repeating itself.
The definition of P_f expresses the following. In the current time slice $P_f(\vec{a})$ produces the data $a_1, \ldots, a_n$ at the output ports $1, \ldots, n$, respectively. In parallel, $P_f(\vec{a})$ waits until one datum is offered at each of the input ports $1, \ldots, m$. The waiting may last into subsequent time slices. When data are available at all input ports, $P_f(\vec{a})$ proceeds with repeating itself from the next time slice with a new output tuple, viz. the value of the function f for the consumed input tuple. The non-deterministic case (P_R) is similar.

For $\mathsf{SProc}(D)$, the operations and constants of BNA as defined on $\mathsf{Proc}(D)$ can be taken with $\mathtt{msd}_1^1$ as wire. This means that only the additional constants for synchronous dataflow have to be defined.

Definition 5.7 (process algebra model for synchronous dataflow)
The operations $+\!\!+$, $\circ$, $\uparrow^n$ on $\mathsf{SProc}(D)$ are the instances of the ones defined on $\mathsf{Proc}(D)$ for $\mathtt{msd}_1^1$ as wire. Analogously, the constants I_n and ${}^m\mathsf{X}^n$ in $\mathsf{SProc}(D)$ are the instances of the ones defined on $\mathsf{Proc}(D)$ for $\mathtt{msd}_1^1$ as wire.

The additional constants in $\mathsf{SProc}(D)$ are defined as follows:

Name	Notation
copy	$\curlywedge^1 \in \mathsf{SProc}(D)(1,2)$
sink	$\multimap^1 \in \mathsf{SProc}(D)(1,0)$
equality test	$\curlyvee_1 \in \mathsf{SProc}(D)(2,1)$
dummy source	$\bullet_1 \in \mathsf{SProc}(D)(0,1)$

Definition

$$\curlywedge^1 \;=\; (1,2,copy^1) \quad \text{where } copy^1 = \underline{\underline{\tau}} \cdot (er_1(x)\ ;\ (\underline{\underline{s}}_1(x) \parallel \underline{\underline{s}}_2(x))) \cdot \sigma_{\mathsf{rel}}(copy^1)$$

$$\multimap^1 \;=\; (1,0,sink^1) \quad \text{where } sink^1 = \underline{\underline{\tau}} \cdot (er_1(x)\ ;\ \underline{\underline{\tau}}) \cdot \sigma_{\mathsf{rel}}(sink^1)$$

$$\curlyvee_1 \;=\; (2,1,eq_1) \quad \text{where } eq_1 = \underline{\underline{\tau}} \cdot (er_1(x_1)\ ;\ P_2(x_1) + er_2(x_2)\ ;\ P_1(x_2))$$
$$\text{and } P_i(x) = \sigma_{\mathsf{rel}}(eq_1) + \underline{\underline{er}}_i(y)\ ;\ (\underline{\underline{s}}_1(x) \lhd x = y \rhd \underline{\underline{\tau}}) \cdot \sigma_{\mathsf{rel}}(eq_1) \text{ for } i \in [2]$$

$$\top_1 \;=\; (0,1,source_1) \quad \text{where } source_1 = \underline{\underline{\tau}} \cdot \delta$$

for $n \neq 1$, these constants are defined by the equations occurring as axioms A12–A19 in Table 3.

□

The equality test $\curlyvee_1$ does not necessarily perform one test per time slice; it does so in order not to cause a time delay. The definition of eq_1 expresses the following. The process eq_1 waits until a datum is offered at one of its input ports. When a datum is available at one input port, it waits till the end of the time slice concerned for a datum at the other port. If this happens, it tests the equality of the data, delivers either in case the test succeeds, and then proceeds with repeating itself from the next time slice. Otherwise, it skips the equality test and proceeds with repeating itself from the next time slice.

The simpler equality test $\overline{\curlyvee}_1 = (2,1,\overline{eq}_1)$, where

$$\overline{eq}_1 = \underline{\underline{\tau}} \cdot ((er_1(x) \parallel er_2(y))\ ;\ (\underline{\underline{s}}_1(x) \lhd x = y \rhd \underline{\underline{\tau}}) \cdot \sigma_{\mathsf{rel}}(\overline{eq}_1))$$

is not appropriate. This equality test does not let data always pass through it with a neglectible delay. This means that it does not behave properly if the feedback operation is applied; $\overline{\curlyvee}_1 \uparrow^1$ is the process that deadlocks after having read one datum – it is a kind of dummy sink. This failure to consume data does not fit in with the idea of permanent flows of data which underlies synchronous dataflow.

Lemma 5.8 *The wire* $\mathsf{I}_1 = (1,1,\mathtt{msd}_1^1)$ *gives an identity flow of data, i.e. for all* $f = (m,n,P)$ *in* $\mathsf{SProc}(D)$, $\mathsf{I}_m \circ f = f = f \circ \mathsf{I}_n$.

Proof: It suffices to show that these equations hold for the atomic cells and the constants. The result then follows by induction on the construction of a network in $\mathsf{SProc}(D)$. $\mathsf{I}_n \circ \mathsf{I}_n = \mathsf{I}_n$ and ${}^m\mathsf{X}^n \circ \mathsf{I}_n = {}^m\mathsf{X}^n = \mathsf{I}_m \circ {}^m\mathsf{X}^n$ follow trivially from $\mathsf{I}_1 \circ \mathsf{I}_1 = \mathsf{I}_1$. For a proof of $\mathsf{I}_1 \circ \mathsf{I}_1 = \mathsf{I}_1$, we refer to [7]. So the asserted equations hold for I_n and ${}^m\mathsf{X}^n$. The proof for the remaining constants and the atomic cells is a laborious piece of work in the same style. □

Theorem 5.9 $(\mathsf{SProc}(D), +\!\!+, \circ, \uparrow, \mathsf{I}, \mathsf{X})$ *is a model of BNA. The constants* $\curlywedge, \multimap, \curlyvee, \top$ *satisfy the additional axioms for synchronous dataflow networks (Table 3).*

Proof: A simple calculation shows that $\mathsf{I}_0 +\!\!+ f = f = f +\!\!+ \mathsf{I}_0$ for all $f \in \mathsf{SProc}(D)$. The first part then follows immediately from Theorem 4.4 and Lemma 5.8. The proof of the second part is a matter of tedious, but unproblematic calculation in the style of [7] (see also the remark after Theorem 6.4). □

Theorem 5.10 *The axioms in Table 3 are complete for closed terms.*

Proof: For the proof of this theorem, we refer to [15]. □

Queues that deliver data with a neglectible delay and never contain more than one datum are an idealized concept; they do not occur in practice. More practical are wires that are interpreted as bounded queues. It seems that bounded queues are most easily modelled as components of asynchronous dataflow networks.

6 Asynchronous dataflow networks

In this section, an extension of BNA for asynchronous dataflow networks is presented. In the first place, the additional constants and axioms for asynchronous dataflow are given. After that, the specialization of the process algebra model of Section 4.3 for asynchronous dataflow networks is described. The adaptation of the data transformer model of Section 4.2 to asynchronous dataflow networks is not described here. Instead, the problem with this model and its proposed solutions are outlined. In Section 7.1, a stream transformer model for the asynchronous case is derived from the process algebra model described in this section – some related models that have been proposed as alternatives are derived there as well.

Various models for asynchronous dataflow have been proposed (see also Section 7) and the valid axioms differ from one model to another. The axioms given here are valid in the presented process algebra model for asynchronous dataflow in case the split/merge interpretation is used for the branching connections. We stress here on the point that we do not present axioms valid in all proposed models for asynchronous dataflow. Neither do we claim completeness with respect to the presented model.

6.1 Additional constants and axioms

The signature of the extension of BNA for asynchronous dataflow networks is obtained by extending the signature of BNA as follows with additional constants for branching connections:

Name	Symbol	Arity
Additional constants:		
split	$\overset{\bullet}{\wedge}^m$	$m \to 2m$
sink	$\underset{\circ}{\mid}^m$	$m \to 0$
merge	$\underset{\bullet}{\vee}_m$	$2m \to m$
dummy source	$\overset{\bullet}{\mid}_m$	$0 \to m$
asynchronous copy	$\overset{\circ}{\wedge}^m$	$m \to 2m$
asynchronous equality test	$\underset{\circ}{\vee}_m$	$2m \to m$

The symbols $\overset{\bullet}{\wedge}^m$, $\underset{\bullet}{\vee}_m$, indicating the split/merge interpretation, as well as the symbols $\overset{\circ}{\wedge}^m$ and $\underset{\circ}{\vee}_m$, indicating the copy/equality test interpretation, are used here. Although the former interpretation seems more close to asynchronous dataflow than the latter interpretation, both are found in asynchronous dataflow.

In Table 4, axioms for the additional constants $\curlywedge^m$, $\bot^m$, $\curlyvee_m$ and $\top_m$ are given. These axioms agree with those for the additional constants of the algebra of flowno-

A1	$(\curlyvee_m \mathbin{++} \mathsf{I}_m) \circ \curlyvee_m = (\mathsf{I}_m \mathbin{++} \curlyvee_m) \circ \curlyvee_m$	A5	(*)
A2	${}^m\mathsf{X}^m \circ \curlyvee_m = \curlyvee_m$	A6	$\curlywedge^m \circ {}^m\mathsf{X}^m = \curlywedge^m$
A3	$(\top_m \mathbin{++} \mathsf{I}_m) \circ \curlyvee_m = \mathsf{I}_m$	A7	(*)
A4	$\curlyvee_m \circ \bot^m = \bot^m \mathbin{++} \bot^m$	A8	$\top_m \circ \curlywedge^m = \top_m \mathbin{++} \top_m$
A9	$\top_m \circ \bot^m = \mathsf{I}_0$		
A10	(*)		
A11	(*)		
A12	$\top_0 = \mathsf{I}_0$	A16	$\bot^0 = \mathsf{I}_0$
A13	$\top_{m+n} = \top_m \mathbin{++} \top_n$	A17	$\bot^{m+n} = \bot^m \mathbin{++} \bot^n$
A14	$\curlyvee_0 = \mathsf{I}_0$	A18	$\curlywedge^0 = \mathsf{I}_0$
A15	$\curlyvee_{m+n} = (\mathsf{I}_m \mathbin{++} {}^n\mathsf{X}^m \mathbin{++} \mathsf{I}_n) \circ (\curlyvee_m \mathbin{++} \curlyvee_n)$	A19	$\curlywedge^{m+n} = (\curlywedge^m \mathbin{++} \curlywedge^n) \circ (\mathsf{I}_m \mathbin{++} {}^m\mathsf{X}^n \mathbin{++} \mathsf{I}_n)$
F3	$\curlyvee_m \uparrow^m = \bot^m$	F4	$\curlywedge^m \uparrow^m = \top_m$
F5	(*)		

Table 4: Additional axioms for asynchronous dataflow networks

mials (Table 2) with five exceptions: A5, A7, A10, A11 and F5 – they all concern the split constant. We consider the axioms in Table 4 desired axioms for asynchronous dataflow networks. They are all valid in the process algebra model described below, but not in some other models. For example, axiom A3 is not valid in Broy's oracle based models [19] (but A5 is valid in these models). The axioms for the constants $\curlywedge$, $\bot$, $\curlyvee$ and $\top$ are the same as the ones in case of synchronous dataflow networks (Table 3).

In the next subsection, the process algebra model introduced in Section 4 is specialized to describe the semantics of the asynchronous dataflow networks.

6.2 Process algebra model for asynchronous dataflow

In this subsection, the specialization of the process algebra model of BNA (Section 4.3) for asynchronous dataflow networks is given. In this case, we will make use of ACP$^\tau$. Recall that ACP$^\tau$ is ACP extended with abstraction based on branching bisimulation.

In Section 4.3, only a few assumption about wires and atomic cells were made. In Section 5.3, these ingredients were actualized for synchronous dataflow networks. Here it is explained how they are actualized for asynchronous dataflow networks. Different from the synchronous case, discrete-time process algebra is not needed.

Definition 6.1 (wires and atomic cells in asynchronous dataflow networks)
In the asynchronous case, the identity constant, now called the *stream delayer*, is the wire $\mathsf{I}_1 = (1, 1, \mathtt{sd}^1_1(\varepsilon))$, where $\mathtt{sd}^1_1$ is defined by

$$\mathtt{sd}^1_1(\sigma) = er_1(x) \,;\, \mathtt{sd}^1_1(\sigma x) + |\sigma| > 0 :\to s_1(hd(\sigma)) \cdot \mathtt{sd}^1_1(tl(\sigma))$$

The constants I_n, for $n \neq 1$, and ${}^m\mathsf{X}^n$ are defined by the equations occurring as axioms B6 and B8–B9, respectively, of Table 1.

In the asynchronous case, the deterministic cell computing a function $f : D^m \to D^n$ is the network $C_f = \mathsf{I}_m \circ (m, n, P_f) \circ \mathsf{I}_n$ where P_f is defined by

$$P_f \;=\; ((er_1(x_1) \parallel \ldots \parallel er_m(x_m)) \,;\, s_1(f_1(x_1, \ldots, x_m)) \parallel \ldots \parallel s_n(f_n(x_1, \ldots, x_m))) * \delta$$

where, for $i \in [n]$, $f_i(x_1, \ldots, x_m) = y_i$ if $f(x_1, \ldots, x_m) = (y_1, \ldots, y_n)$.

The non-deterministic cell computing a (finitely branching) relation $R \subseteq D^m \times D^n$ is the network $C_R = \mathsf{I}_m \circ (m, n, P_R) \circ \mathsf{I}_n$ where P_R is defined by

$$\begin{aligned} P_R \;=\; & ((er_1(x_1) \parallel \ldots \parallel er_m(x_m)) \,; \\ & \qquad \tau \lhd R(x_1, \ldots, x_m) = \emptyset \rhd \sum_{(a_1,\ldots,a_n) \in R(x_1,\ldots,x_m)} (s_1(a_1) \parallel \ldots \parallel s_n(a_n))) * \delta \end{aligned}$$

The restriction of $\mathsf{Proc}(D)$ to the processes that can be built under this actualization is denoted by $\mathsf{AProc}(D)$. $\Box$

The definition of $\mathtt{sd}_1^1$ simply expresses that it behaves as a queue. The definition of P_f expresses the following. P_f waits until one datum is offered at each of the input ports $1, \ldots, m$. When data is available at all input ports, P_f proceeds with producing data at the output ports $1, \ldots, n$. The datum produced at the i-th output port is the i-component of the value of the function f for the consumed input tuple. When data is delivered at all output ports, P_f proceeds with repeating itself.

For $\mathsf{AProc}(D)$, the operations and constants of BNA as defined on $\mathsf{Proc}(D)$ can be taken with $\mathtt{sd}_1^1$ as wire. This means that only the additional constants for asynchronous dataflow have to be defined.

Definition 6.2 (process algebra model for asynchronous dataflow)
The operations $\mathbin{+\!\!+}$, $\circ$, $\uparrow^n$ on $\mathsf{AProc}(D)$ are the instances of the ones defined on $\mathsf{Proc}(D)$ for $\mathtt{sd}_1^1$ as wire. Analogously, the constants I_n and ${}^m\mathsf{X}^n$ in $\mathsf{AProc}(D)$ are the instances of the ones defined on $\mathsf{Proc}(D)$ for $\mathtt{sd}_1^1$ as wire.

The additional constants in $\mathsf{AProc}(D)$ are defined as follows:

Name	Notation
split	$\overset{\bullet}{\wedge}{}^1 \in \mathsf{AProc}(D)(1,2)$
sink	$\underset{\circ}{\vert}{}^1 \in \mathsf{AProc}(D)(1,0)$
merge	$\underset{\bullet}{\vee}{}_1 \in \mathsf{AProc}(D)(2,1)$
dummy source	$\overset{\bullet}{\vert}{}_1 \in \mathsf{AProc}(D)(0,1)$
asynchronous copy	$\overset{\circ}{\wedge}{}^1 \in \mathsf{AProc}(D)(1,2)$
asynchronous equality test	$\underset{\circ}{\vee}{}_1 \in \mathsf{AProc}(D)(2,1)$

Definition

$$\curlywedge^1 = \mathsf{I}_1 \circ (1, 2, split^1) \circ \mathsf{I}_2 \quad \text{where } split^1 = (er_1(x) \ ; (s_1(x) + s_2(x))) * \delta$$

$$\multimap^1 = \mathsf{I}_1 \circ (1, 0, sink^1) \quad \text{where } sink^1 = (er_1(x) \ ; \tau) * \delta$$

$$\curlyvee_1 = \mathsf{I}_2 \circ (2, 1, merge_1) \circ \mathsf{I}_1 \quad \text{where } merge_1 = ((er_1(x) + er_2(x)) \ ; s_1(x)) * \delta$$

$$\top_1 = (0, 1, source_1) \circ \mathsf{I}_1 \quad \text{where } source_1 = \delta$$

$$\wedge^1 = \mathsf{I}_1 \circ (1, 2, acopy^1) \circ \mathsf{I}_2 \quad \text{where } acopy^1 = (er_1(x) \ ; (s_1(x) \parallel s_2(x))) * \delta$$

$$\vee_1 = \mathsf{I}_2 \circ (2, 1, aeq_1) \circ \mathsf{I}_1 \quad \text{where } aeq_1 = ((er_1(x_1) \parallel er_2(x_2)) \ ; s_1(x_1) \lhd x_1 = x_2 \rhd s_1(\surd)) * \delta$$

for $n \neq 1$, these constants are defined by the equations occurring as axioms A12–A19 in Table 4 (for split, sink, merge and dummy sink) or Table 3 (for asynchronous copy and equality test).

□

The asynchronous versions of the constants sink, dummy source and copy given here agree with the synchronous versions given in Section 5.3. The asynchronous version of the equality test does not agree with its synchronous version. It agrees with the simpler equality test ($\overline{\vee}$), also mentioned in Section 5.3, which is not appropriate in the synchronous case. In order to be fully precise, we have to adapt the definitions given in Section 4.3 in the case of asynchronous dataflow with the equality test as additional constant: all occurrences of the condition $d \in D$ have to be replaced by $d \in D \cup \{\surd\}$.

Lemma 6.3 *The wire* $\mathsf{I}_1 = (1, 1, \mathsf{sd}_1^1)$ *gives an identity flow of data, i.e. for all* $f = (m, n, P)$ *in* $\mathsf{AProc}(D)$, $\mathsf{I}_m \circ f = f = f \circ \mathsf{I}_n$.

Proof: For I_1, it is well known that $\mathsf{I}_1 \circ \mathsf{I}_1 = \mathsf{I}_1$ (see e.g. [28]). $\mathsf{I}_n \circ \mathsf{I}_n = \mathsf{I}_n$ and ${}^m\mathsf{X}^n \circ \mathsf{I}_n = {}^m\mathsf{X}^n = \mathsf{I}_m \circ {}^m\mathsf{X}^n$ follow trivially from $\mathsf{I}_1 \circ \mathsf{I}_1 = \mathsf{I}_1$. So the asserted equations hold for I_n and ${}^m\mathsf{X}^n$. Due to the pre- and postfixing with identities in the definitions of the remaining constants and the atomic cells, it follows trivially that these equations hold also for them. The result then follows by induction on the construction of a network in $\mathsf{AProc}(D)$. □

Theorem 6.4 $(\mathsf{AProc}(D), \#, \circ, \uparrow, \mathsf{I}, \mathsf{X})$ *is a model of BNA. The constants* $\curlywedge, \multimap, \curlyvee, \top$ *satisfy the additional axioms for asynchronous dataflow networks (Table 4). The constants* $\wedge, \multimap, \vee, \top$ *satisfy the additional axioms for synchronous dataflow networks (Table 3).*

Proof: A simple calculation shows that $\mathsf{I}_0 \# f = f = f \# \mathsf{I}_0$ for all $f \in \mathsf{AProc}(D)$. The first part then follows immediately from Theorem 4.4 and Lemma 6.3. The proof of the second and third part is a matter of tedious, but unproblematic calculation in the style of, for example, [30, 38]. □

We do not provide a detailed proof of the second and third part of Theorem 6.4 for various reasons. Different strategies for such a proof are possible, but for each of them the proof will turn out to be a long listing of rather uninteresting calculations.

The principal degree of freedom lies in the fraction of formal equational reasoning from axioms versus semantic work directly in the model of process graphs modulo branching bisimulation. A proof within this model will be rather unreadable and still informal. If a proof using equational reasonong is made, one needs a proof system such as the one for μCRL [31] and a systematic use of the conditional alphabet axioms introduced in [8]. This kind of proofs can be found in [30, 38]. There the proofs have been worked out to the level of detail that they can be formalized in the underlying type theory of the proof assistant Coq (see e.g. [25]) and automatically checked. This approach will work as well for the second and third part of Theorem 6.4, i.e. for the network algebra axioms concerned. We have not followed this approach because fully formal proofs would in this case not increase the plausibility of the axioms concerned.

Note that the third part of Theorem 6.4 expresses that the algebraic structure of the synchronous dataflow networks is preserved in the asynchronous setting. That is, the asynchronous dataflow networks built using copy and equality test – instead of split and merge – as branching constants, satisfy the same axioms as the synchronous dataflow networks.

6.3 More abstract models for asynchronous dataflow

Just like for synchronous dataflow networks, a more abstract model based on stream transformers can be given for deterministic asynchronous dataflow networks. A result of Kahn [34] shows that this model is compositional. As shown by Brock and Ackermann [18], and Keller [35], the model is not compositional in the nondeterministic case. Some networks that are equivalent – realize the same relation between their input and output streams – can not be substituted for each other in a larger network because equivalence will get lost.

This deviation, known as the Brock-Ackermann anomaly and the merge anomaly, is a time anomaly. It is related to the feedback operation. Consider an arbitrary deterministic dataflow network with a feedback loop. If the network gets data faster from its feedback loop, the additional data do not change at any moment the prefix of the streams being produced because the network is deterministic. So only the relation between the input and output streams matters. However, in the nondeterministic case, the timing differences in producing the data that is fed back become important. The Brock-Ackermann example relies on such timing differences to show that the feedback of certain networks with the same relation between its input and output streams are different.

One may try to solve the anomaly in two ways:

(1) weaken the abstract model,
(2) strengthen the operational model.

On the lines of (1), several models have been proposed [18, 19, 33, 37]. The general approach of these proposals can be described as follows: add more detail to the model, but keep unchanged the operational interpretation of wires as unbounded queues. In this way the simple stream transformer model is sacrificed and other models emerge: trace models giving global time information by merging all the local streams into one trace, oracle based models reducing nondeterministic behaviour to deterministic behaviour up to certain oracles and using the compositionality of the stream transformer model for deterministic dataflow networks, etc. The stream transformer model and some of these more detailed models are the subject of Section 7. The stream transformer model for asynchronous dataflow is commonly referred to as the history model.

7 Related models for asynchronous dataflow

In this section, several different models for asynchronous dataflow are explained from the angle of the process algebra model presented above. First of all, Kahn's history model [34], Broy's oracle based models [19] and Jonsson's trace model [33] are derived from the process algebra model presented in Section 6.2. Next, the time anomaly, which may occur in the history model, is explained using the Brock-Ackermann example. After that, the derived models are broadly compared with each other. Finally, a different process algebra model, based on guess-and-borrow queues, is outlined. With this new operational model, the time anomaly disappears.

7.1 Derivation of related models

In this subsection, the derivation of several models from the process algebra model for asynchronous dataflow is described. Connecting the process algebra model with the history model and the oracle based models, requires a somewhat unnatural reconstruction of these models. We provide a description of it below, but we agree that it does not go smoothly.

In asynchronous dataflow, consumption and production of data is not driven by clock ticks. This means that it is unnatural to uphold the view-point concerning streams taken in Section 5.2 for synchronous dataflow. More precisely, in this section a stream is considered to be an element of D^ω, i.e. a finite or infinite sequence of data.

In order to be able to use well-known models of process algebra in the derivation of the history model, and the oracle based models using the history model, the input streams of a network have to be represented by networks. These input networks are then composed with the original network.

Definition 7.1 (input network)
Let σ be a stream. The input network associated with σ is the network $\mathtt{SOURCE}_1(\sigma) = (0, 1, \tau \cdot \mathtt{source}_1(\sigma))$ where

$$\mathtt{source}_1(\sigma) \;=\; |\sigma| > 0 :\rightarrow s_1(hd(\sigma)) \cdot \mathtt{source}_1(tl(\sigma))$$

Let $f : m \to n$ be a network and $\sigma_1, \ldots, \sigma_m$ be streams. The network $f(\sigma_1, \ldots, \sigma_m)$ is defined by

$$f(\sigma_1, \ldots, \sigma_m) = (\mathtt{SOURCE}_1(\sigma_1) \mathbin{+\!\!+} \ldots \mathbin{+\!\!+} \mathtt{SOURCE}_1(\sigma_m)) \circ f$$

□

For given input streams, the output streams can be reconstructed from the complete traces of the process corresponding to the composed network as described above. We write $\mathsf{trace}(P)$, where P is a process, for the set of complete traces of P. What we are talking about here is the union of the complete traces of P as defined in [14], the traces of P that become complete if we identify livelock nodes (i.e. nodes that only permit an infinite path of silent steps) with deadlock nodes, and the infinite traces of P. $\mathsf{trace}(P)$ is formally defined in [27], where it is called the set of fair traces of P. Note however that the distinction between successful termination and deadlock/livelock made in such traces is irrelevant here because the processes modeling asynchronous dataflow networks do not include successfully terminating processes.

Definition 7.2 (stream extraction)
Let β be a trace over $\{s_i(d) \mid i \in [m], d \in D\} \cup \{r_j(d) \mid j \in [n], d \in D\}$. We write $\mathsf{stream}_i^{in}(\beta)$ for the stream of data obtained by first removing all actions that are not of the form $r_i(d)$ and after that replacing each action of the form $r_i(d)$ by d. Analogously, we write $\mathsf{stream}_i^{out}(\beta)$ for the stream of data obtained by first removing all actions that are not of the form $s_i(d)$ and after that replacing each action of the form $s_i(d)$ by d. $\Box$

For a network $f : m \to n$ and an m-tuple of streams $(\sigma_1, \ldots, \sigma_m)$, the possible n-tuples of output streams can now be obtained from the traces of the process corresponding to the network $f(\sigma_1, \ldots, \sigma_m)$ using stream extraction.

Definition 7.3 (history relation)
We write $\mathsf{trace}(f)$, where $f = (m, n, P)$ is a network, for $\mathsf{trace}(P)$. The input-output *history relation* of a network $f : m \to n$, written $[f]$, is defined by

$$[f](\sigma_1, \ldots, \sigma_m) \;=\; \{(\mathsf{stream}_1^{out}(\beta), \ldots, \mathsf{stream}_n^{out}(\beta)) \mid \beta \in \mathsf{trace}(f(\sigma_1, \ldots, \sigma_m))\}$$

$\Box$

The associated equivalence on networks corresponds to Kahn's history model [34]. Hence the following definition.

Definition 7.4 ($\equiv_{\text{history}}$)
The *history* equivalence $\equiv_{\text{history}}$ on asynchronous dataflow networks is defined by $f \equiv_{\text{history}} g$ iff $[f] = [g]$. $\Box$

Broy's oracle based models [19], which are closely related to Kahn's history model, may be derived from the process algebra model as well. To this end we consider the following merge and split using oracles.

Definition 7.5 (split and merge with oracles)
Let $\alpha \in \{1, 2\}^\infty$ be an oracle. The split and merge constants with oracles are defined on $\mathsf{AProc}(D)$ as follows:

Name	Notation	
split with oracle	$\curlywedge^1(\alpha)$	$\in \mathsf{AProc}(D)(1, 2)$
merge with oracle	$\curlyvee_1(\alpha)$	$\in \mathsf{AProc}(D)(2, 1)$

Definition

$$\curlywedge^1(\alpha) \;=\; \mathsf{l}_1 \circ (1, 2, split^1(\alpha, 0)) \circ \mathsf{l}_2$$
$$\text{where } split_1(\alpha, i) = (er_1(x)\ ;\ s_1(x) \lhd \alpha(i) = 1 \rhd s_2(x)) \cdot split_1(\alpha, i+1)$$

$$\curlyvee_1(\alpha) \;=\; \mathsf{l}_2 \circ (2, 1, merge_1(\alpha, 0)) \circ \mathsf{l}_1$$
$$\text{where } merge_1(\alpha, i) = (er_1(x) \lhd \alpha(i) = 1 \rhd er_2(x)\ ;\ s_1(x)) \cdot merge_1(\alpha, i+1)$$

□

Definition 7.6 ($\equiv_{\text{broy}}$ and $\equiv_{\text{broy-fair}}$)
Let $\alpha_1, \ldots, \alpha_k \in \{1,2\}^\infty$ be oracles and let $f(\alpha_1 \ldots \alpha_k)$ be the network obtained form f by replacing each occurrence of $\curlywedge^1$ and $\curlyvee_1$ by $\curlywedge^1(\alpha_i)$ and $\curlyvee_1(\alpha_i)$, respectively, where i is a unique index for the occurrence concerned in f.

Let $f, g : m \to n$ be networks, and let $1, \ldots, k$ and $1, \ldots, l$ be the indices for the occurrences of $\curlywedge^1$ and $\curlyvee_1$ in f and g, respectively. f and g are *Broy equivalent*, written $f \equiv_{\text{broy}} g$, iff for all oracles $\alpha_1, \ldots, \alpha_k \in \{1,2\}^\infty$, there exists oracles $\beta_1, \ldots, \beta_l \in \{1,2\}^\infty$ such that

$$(*) \qquad [f(\alpha_1, \ldots, \alpha_k)] = [g(\beta_1, \ldots, \beta_l)]$$

holds and reverse, for all oracles $\beta_1, \ldots, \beta_l \in \{1,2\}^\infty$, there exists oracles $\alpha_1, \ldots, \alpha_k \in \{1,2\}^\infty$ such that $(*)$ holds.

f and g are *Broy-fair equivalent*, written $f \equiv_{\text{broy-fair}} g$, iff $f \equiv_{\text{broy}} g$ and all α's and β's are fair. An oracle $\alpha \in \{1,2\}^\infty$ is fair iff $|\alpha^{-1}(1)| = |\alpha^{-1}(2)|$.

In Section 7.3, we will write $[f]_{\text{broy}}(\sigma_1, \ldots, \sigma_m)$ for $\bigcup\{[f(\alpha_1, \ldots, \alpha_k)](\sigma_1, \ldots, \sigma_m) \mid \alpha_1, \ldots, \alpha_k \in \{1,2\}^\infty\}$. □

Various interesting models for process algebra are obtained by defining equivalence relations on process graphs. For a systematic treatment of most of these equivalence relations, the reader is referred to [9]. We mention:

$\equiv_{\text{ct}}$	completed trace equivalence,
$\leftrightarrows_{\text{w}}$	weak bisimulation equivalence,
$\leftrightarrows_{\text{b}}$	branching bisimulation equivalence.

Weak and branching bisimulation were introduced, in the setting of ACP, in [13] and [29], respectively. $P \equiv_{\text{ct}} Q$ iff $\mathsf{trace}(P) = \mathsf{trace}(Q)$. The above-mentioned equivalences on process graphs naturally induce corresponding equivalences on asynchronous dataflow networks. For example, the equivalence induced by $\equiv_{\text{ct}}$ corresponds to Jonsson's trace model [33].

Definition 7.7 ($\equiv_{\text{trace}}$)
Let $f = (m, n, P)$ and $g = (p, q, Q)$ be two networks. f and g are *trace equivalent*, written $f \equiv_{\text{trace}} g$, iff $m = p$, $n = q$ and $P \equiv_{\text{ct}} Q$. □

Another interesting equivalence on asynchronous dataflow networks induced by the above-mentioned equivalences on process graphs is the following.

Definition 7.8 ($\equiv_{\text{bisim}}$)
Let $f = (m, n, P)$ and $g = (p, q, Q)$ be two networks. f and g are *bisimulation equivalent*, written $f \equiv_{\text{bisim}} g$, iff $m = p$, $n = q$ and $P \leftrightarrows_{\text{b}} Q$. □

After the next subsection, which explains the time anomaly in the history model, the models derived in this subsection are related to each other.

7.2 Time Anomaly

In this subsection, the time anomaly is illustrated by means of two examples: the Brock-Ackermann example [18] and an example originating from Russell [40].

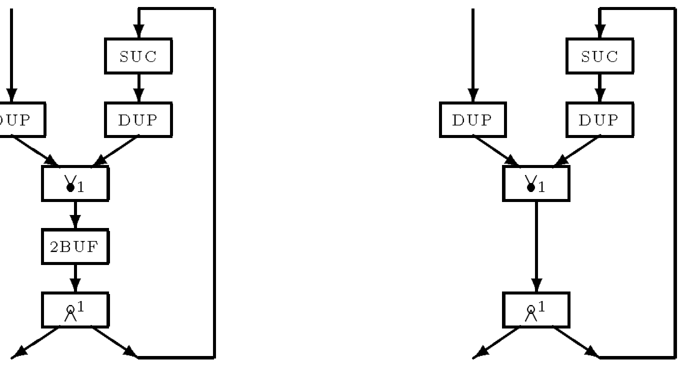
Example 7.9 (Brock-Ackermann example)
The Brock-Ackermann example is depicted in Figure 4. Here $\curlyvee_1$ and $\curlywedge^1$ are the merge and copy constants for asynchronous dataflow networks defined in Section 6.2. The atomic cells used in this example are:

$$\begin{array}{lcl}
\mathtt{SUC} & = & (1,1,(er_1(x)\ ;\ s_1(x+1))\ {}^*\ \delta) \\
\mathtt{DUP} & = & (1,1,(er_1(x)\ ;\ s_1(x)\cdot s_1(x))\ {}^*\ \delta) \\
\mathtt{2BUF} & = & (1,1,((er_1(x)\cdot er_1(y))\ ;\ (s_1(x)\cdot s_1(y)))\ {}^*\ \delta)
\end{array}$$

The following networks are built from these atomic cells:

$$\begin{array}{lcl}
f & = & (\mathtt{DUP} \mathbin{+\!\!+} \mathtt{SUC} \circ \mathtt{DUP}) \circ \curlyvee_1 \circ \mathtt{2BUF} \circ \curlywedge^1 \\
f' & = & (\mathtt{DUP} \mathbin{+\!\!+} \mathtt{SUC} \circ \mathtt{DUP}) \circ \curlyvee_1 \circ \mathsf{I}_1 \circ \curlywedge^1
\end{array}$$

It is easy to see that the networks f and f' realize the same relation between their input and output streams, i.e. $f \equiv_{\text{history}} f'$ (notice that $\mathtt{2BUF} \equiv_{\text{history}} \mathsf{I}_1$). However, f and f' can not always be substituted for each other in a larger network. Consider, for instance, the networks $f \uparrow^1$ and $f' \uparrow^1$. The stream $1223\ldots$ is in $[f' \uparrow^1](1)$ but not in $[f \uparrow^1](1)$ because 2BUF must have consumed both 1's yielded by the duplication of the input before the feedback loop can contribute to the output. So $f \uparrow^1 \not\equiv_{\text{history}} f' \uparrow^1$. $\square$



$f \uparrow^1$ $\qquad$ $f' \uparrow^1$

Figure 4: Brock-Ackermann example

Example 7.10 (Russell's example)
In the previous example, the atomic cells were all deterministic. The merge constant introduced nondeterminism in that example. Russell's example shows that the time anomaly may also occur by nondeterministic atomic cells. Consider the following atomic cells:

$$\begin{array}{lcl}
g & = & (1,1,((er_1(x)\ ;\ s_1(0)\cdot s_1(1)) + s_1(0)\cdot(er_1(x)\ ;\ s_1(0)))\ {}^*\ \delta) \\
g' & = & (1,1,((er_1(x)\ ;\ s_1(0)\cdot s_1(1)) + s_1(0)\cdot(er_1(x)\ ;\ s_1(0)) + \\
 & & \qquad s_1(0)\cdot(er_1(x)\ ;\ s_1(1)))\ {}^*\ \delta)
\end{array}$$

It is easy to see that the atomic cells g and g' realize the same relation between their input and output streams, i.e. $g \equiv_{\text{history}} g'$. Note further that the possible output streams are independent of the input streams. However, the stream $01\ldots$ is contained in $[(g' \circ \Re^1) \uparrow^1]()$ but not in $[(g \circ \Re^1) \uparrow^1]()$, because g must already have produced one datum before it can contribute a 0 followed by a 1 to the output. So $(g \circ \Re^1) \uparrow^1 \not\equiv_{\text{history}} (g' \circ \Re^1) \uparrow^1$. $\Box$

7.3 Comparison of models

We have defined the following equivalences on networks in the process algebra model for asynchronous dataflow:

$\equiv_{\text{history}}$	history equivalence,
$\equiv_{\text{broy}}$	Broy equivalence,
$\equiv_{\text{broy-fair}}$	Broy-fair equivalence,
$\equiv_{\text{trace}}$	trace equivalence,
$\equiv_{\text{bisim}}$	bisimulation equivalence.

The process algebra model modulo the first four equivalences yields Kahn's history model [34], Broy's oracle-based models [19] and Jonsson's trace model [33], respectively. Thus, we know from the relevant literature that the first of these equivalences is not a congruence (in [18] is shown that the history model is not compositional) and that the others are congruences (in [19] and [33] is shown that the Broy models and the trace model are compositional). In [33], it is further shown that the trace model is fully abstract with respect to the history model. The bisimulation equivalence corresponds to the process algebra model itself – where processes are considered to be equal iff they are branching bisimulation equivalent. The following summarizes how the above-mentioned equivalences are related (thus showing the connections between the various models):

$$\equiv_{\text{bisim}} \overset{(1)}{\underset{\not\supseteq}{\not\subseteq}} \equiv_{\text{broy}} \overset{(2)}{\underset{\not\supseteq}{\not\subseteq}} \equiv_{\text{broy-fair}}, \quad \equiv_{\text{bisim}} \overset{(3)}{\subset} \equiv_{\text{trace}}, \quad \equiv_{\text{broy}} \overset{(4)}{\subset} \equiv_{\text{trace}} \overset{(5)}{\subset} \equiv_{\text{history}}$$

The incomparibilities (1) and (2) are shown in Example 7.11 and Example 7.12, respectively. The inclusion (3) follows trivially from the inclusion of the corresponding equivalences for processes (see e.g. [27]). It is obvious from the definition of $\equiv_{\text{broy}}$ that $\equiv_{\text{broy}} \subseteq \equiv_{\text{history}}$. Because of the compositionality of the Broy model and the full abstractness of the trace model, the inclusion (4) follows then immediately, except for its strictness. In Example 7.13 is shown that the inclusion (4) is strict. The full abstractness of the trace model with respect to the history model and the non-compositionality of the history model entail directly the strict inclusion (5). The proofs below are quite sketchy. In the case of equivalences like $\equiv_{\text{broy}}$ in Example 7.11, such statements require a further formal proof using invariants that we have not included. For use of invariants in the process algebra setting, we refer to [16, 32].

Example 7.11 ($\equiv_{\text{broy}}$ and $\equiv_{\text{bisim}}$ are incomparable)
First, we give an example of two networks which are bisimulation equivalent, but not Broy equivalent. The following atomic cells occur in the networks concerned:

$$\begin{array}{rcl} \texttt{SOURCE}(i) & = & (0,1,\tau\cdot(s_1(i)\ ^{*}\ \delta)) \\ \texttt{FILTER\_0} & = & (1,1,\tau\cdot(er_1(x)\ ;\ (s_1(0) \lhd x=0 \rhd \tau))\ ^{*}\ \delta) \end{array}$$

Let $f = \texttt{SOURCE}(0)$ and $g = (\texttt{SOURCE}(0) \mathbin{+\!\!+} \texttt{SOURCE}(1)) \circ \curlyvee_1 \circ \texttt{FILTER\_0}$. Then (i) $f \not\equiv_{\text{broy}} g$ and (ii) $f \equiv_{\text{bisim}} g$. For (i), we see that $[f]_{\text{broy}}() = \{0^\infty\}$ and $[g]_{\text{broy}}() = \{0^\infty, \varepsilon\}$ (g produces the ε output for the completely unfair oracle $\alpha = 2^\infty$). For (ii), notice that the corresponding processes are branching bisimilar because: (a) in each state after a number of τs an output is generated and (b) no other outputs may be generated.

Next, we give an example of networks which are Broy equivalent, but not bisimulation equivalent. Let $f = \curlywedge^1 \circ (\mathsf{I}_1 \mathbin{+\!\!+} \curlywedge^1)$ and $g = \curlywedge^1 \circ (\curlywedge^1 \mathbin{+\!\!+} \mathsf{I}_1)$. Then (i) $f \equiv_{\text{broy}} g$ and (ii) $f \not\equiv_{\text{bisim}} g$. For (i), we see that for each pair of oracles for f one may find a pair of oracles for g which produces the same output stream as f and conversely. For (ii), we see that the branching structure of f and g differ. $\square$

Example 7.12 ($\equiv_{\text{broy}}$ and $\equiv_{\text{broy-fair}}$ are incomparable)
The first example from 7.11 provides two networks which are Broy-fair equivalent, but not Broy equivalent. For the reversed non-inclusion, we use an additional atomic cell $\texttt{STOP\_AT\_1} = (1,1,\texttt{stop\_at\_1})$ where

$$\texttt{stop\_at\_1} = \tau\cdot(r_1(0)\cdot s_1(0)\cdot \texttt{stop\_at\_1} + r_1(1)\cdot(er_1(x)\ ;\ \tau)\ ^{*}\ \delta)$$

Let $f = \texttt{SOURCE}(0) \circ \curlywedge^1 \circ (\flat^1 \mathbin{+\!\!+} \mathsf{I}_1)$ and $g = (\texttt{SOURCE}(0) \mathbin{+\!\!+} \texttt{SOURCE}(1)) \circ \curlyvee_1 \circ \texttt{STOP\_AT\_1}$. Then $f \equiv_{\text{broy}} g$, but $f \not\equiv_{\text{broy-fair}} g$. $\square$

The following example conforms the remark in [20] that $\equiv_{\text{broy}}$ is not fully abstract, in view of the fact that $\equiv_{\text{trace}}$ is fully abstract, cf. e.g. [33].

Example 7.13 ($\equiv_{\text{broy}}$ is strictly included in $\equiv_{\text{trace}}$)
Let $f = \curlywedge^1 \circ \curlyvee_1$ and $f' = \curlywedge^1 \circ (\texttt{2BUF} \mathbin{+\!\!+} f) \circ \curlyvee_1$ ($\texttt{2BUF}$ is the component that appears in Example 7.9). Then, for $D = \{0,1\}$, (i) $f \not\equiv_{\text{broy}} f'$ and (ii) $f \equiv_{\text{trace}} f'$. For f we need two oracles $\alpha_s, \alpha_m \in \{1,2\}^\infty$ for the split and the merge components, respectively; and for f' we need two more oracles $\alpha'_s, \alpha'_m \in \{1,2\}^\infty$ for the additional components. For (i), we see that the function computed by f' for oracles α'_s and α'_m with the prefix 11 can not be computed by f for any two oracles α_s and α_m. Suppose the contrary. Then there are two oracles $\alpha_s = s_0 s_1 s_2 \ldots$ and $\alpha_m = m_0 m_1 m_2 \ldots$ such that $[f(\alpha_s,\alpha_m)](0) = \{\varepsilon\}$ and $[f(\alpha_s,\alpha_m)](01) = \{01\}$. Since $[f(\alpha_s,\alpha_m)](0) = \{\varepsilon\}$, $s_0 \neq m_0$. So for the input 01 two cases are left, $s_1 \neq m_0$ and $s_1 = m_0$, which both lead to contradiction: $[f(\alpha_s,\alpha_m)](01) = \{\varepsilon\}$ if $s_1 \neq m_0$, and $[f(\alpha_s,\alpha_m)](01) = \{1\}$ or $[f(\alpha_s,\alpha_m)](01) = \{10\}$ (depending on m_1) if $s_1 = m_0$. For (ii), we see immediately that each trace of f is a trace of f' as well. And conversely, we see that every trace w of f' has the property that, for all $d \in D$ and $n \in \mathbb{N}$, $card\{i \mid i \leq n, w_i = s_1(d)\} \leq card\{i \mid i \leq n, w_i = r_1(d)\}$. Now, every trace with this property is a trace of f because: (a) the split component may deliver all 0s to the "left" and all 1s to the "right" and (b) the merge component may always consume from the left if a 0 is necessary to produce a trace with the property above and from the right if a 1 is necessary. $\square$

Note the following. Let $\alpha_s = (12)^\infty$ and $\alpha_m = (21)^\infty$ be oracles for the split connection and the merge connection of the network f in Example 7.13 above. Then $[f(\alpha_s,\alpha_m)](d_2 d_1 d_4 d_3 \ldots) = [\texttt{2BUF}](d_1 d_2 d_3 d_4 \ldots)$. Hence, if D has only one element, e.g. $D = \{0\}$, then f and f' are Broy (and Broy-fair) equivalent.

7.4 Guess-and-borrow queues

Different from a synchrononous dataflow network, an asynchronous one may delay the use of its resources. In case asynchronism is exploited fully, it should also be possible to use the resources in advance.

In the Brock-Ackermann example (Example 7.9) $[f' \uparrow^1](1)$ contains $1223\ldots$, which is not in $[f \uparrow^1](1)$ under the interpretation of dataflow networks where wires are treated as unbounded queues. With a more powerful kind of identity connections, viz. guess-and-borrow queues, this anomaly disappears.

A *guess-and-borrow queue* may deliver any number of arbitrary data to a cell while it is empty. However, if this turns out not to be in agreement with the actual data subsequently received, the cell has to drop the computation based on the wrong data.

In the Brock-Ackermann example, $1223\ldots$ is a common output stream of $f \uparrow^1$ and $f' \uparrow^1$ if the identity connections are interpreted as (unbounded) guess-and-borrow queues. One only needs such a queue before the `2BUF` cell; which now may borrow the necessary data in order to annihilate the differences between the `2BUF` cell and an identity connection. More generally, one may see that $f \uparrow^1$ and $f' \uparrow^1$ compute the same input-output relation on streams with this more powerful operational interpretation for the identity connections. A similar simple argument works in the case of Russell's example as well.

We now briefly outline the construction of a process algebra model which can be regarded as the operational model of asynchronous dataflow networks with guess-and-borrow queues. In the style of Parrow [39], we use as processes pairs (p, U) with p a process modulo $\leftrightarrows_{b\Delta}$ (divergence sensitive branching bisimulation)[2] and U a collection of admissible complete traces for p. The process algebra operators are to be extended to the trace set component. This is straightforward, except for the point that in $X \parallel Y$ complete traces must be formed by merging complete traces for both X and Y in such a way that all actions are "used". We also define $\mathsf{trace}(p, U) = \mathsf{trace}(p) \cap U$. With these ingredients we may give a process algebra definition of the identity constant as a guess-and-borrow queue as follows. The identity constant, called *stream retimer*, is the wire $\mathsf{I}_1 = (1, 1, \mathtt{sr}_1^1)$, where $\mathtt{sr}_1^1 = (p, U)$ and

$$p = (\sum_{d \in D} r_1(d) + \sum_{d \in D} s_1(d)) * \delta$$

$$U = \{\alpha \mid \alpha \text{ is complete trace over } \{r_1(d), s_1(d) \mid d \in D\}, \mathsf{stream}_1^{in}(\alpha) = \mathsf{stream}_1^{out}(\alpha)\}$$

It can be shown that $\mathsf{I}_1 \circ \mathsf{I}_1 = \mathsf{I}_1$, so I_1 is an identity flow. However, this identity flow allows to shift a stream forward and backward in time.

The feedback operation in the outlined operational model of asynchronous dataflow networks with guess-and-borrow queues corresponds to the greatest fixpoint approach – applied in Definition 4.1 – in the stream transformer model for this kind of dataflow. The equation $\curlywedge^m \uparrow^m = \text{♀}_m$, where $\text{♀}_n = \{\langle (), x\rangle \mid x \in S^n\}$, now holds.

[2]In case of divergence sensitive bisimulation, if two nodes are related by a bisimulation and one node permits an infinite path of silent steps, the other node must also permit an infinite path of silent steps. Divergence sensitive bisimulation was introduced in [14].

8 Closing remarks

Concerning connections with earlier work on dataflow some additional remarks are in order.

In [10] a model for synchronous dataflow networks is presented. Our Section 5.2 on a stream transformer model for synchronous dataflow can be seen as a rephrasing of this work. We consider the stream transformer model described in Section 5.2 to be more denotational and the process algebra model described in Section 5.3 to be more operational.

The model presented in [10] is essentially a BNA model, although it has some slightly different operations and constants. For example, it has "left-feedback" (*) instead of "right-feedback" (see also the table below) and "input sharing" (^) instead of the constants $\mathring{\wedge}$ and X. However, the constants and operations of BNA are definable in terms of the ones of this model and vice versa. The setting of [10] may be obtained from our general network algebra setting by taking BNA with the following parameters: (1) the set of data D is $\mathbb{N}$; (2) the atomic cells are "successor" and "conditional"; (3) the additional constants for branching connections are $\mathring{\wedge}$, ⫰ and $\mathring{\vee}$. Kahn's history model [34] is also essentially a BNA model (with $\mathring{\wedge}$, ⫰ and ⫯ as additional constants) and so are Broy's oracle based models [19].

The SCAs [44] require for each internal stream in a network an initial value. We have taken that viewpoint as well and this leads to a major distinction between our process algebra models for synchronous and asynchronous dataflow networks.

Both the left- and right-feedback can be used. The left-feedback can be defined in terms of the right-feedback as follows:

$$\uparrow^p (f) = ({}^p\mathsf{X}^m \circ f \circ {}^p\mathsf{X}^n) \uparrow^p, \quad f : p + m \to p + n$$

Other proposed feedback-like operators can be defined in terms of left- or right-feedback:

Symbol	Name	Network algebra specification	In
$*$	feedback	$f^* = \uparrow^1 f, \ f : 1 + m \to 1 + n$	[10]
μ	feedback	$\mu f = (f \circ \wedge^m) \uparrow^m, \ f : n + m \to m$	[20]
$*$	(unary) star	$f^* = \wedge^1 \circ (\mathsf{I}_1 \mathbin{+\!\!\!+} (\vee_1 \circ f \circ \wedge^1) \uparrow^1) \circ \vee_1, \ f : 1 \to 1$	[24]
$\dagger$	iteration	$f^\dagger = \uparrow^m (\vee_m \circ f), \ f : m \to m + n$	[26]
$*$	(binary) star	$f^* g = \wedge^1 \circ (\mathsf{I}_1 \mathbin{+\!\!\!+} \uparrow^1 (\vee_1 \circ f \circ \wedge^1)) \circ \vee_1 \circ g, \ f, g : 1 \to 1$	[36]

Acknowledgements

The understanding on dataflow computation of the third author was much clarified by discussions with M. Broy and K. Stølen. The first author acknowledges discussions with J.V. Tucker on SCAs.

References

[1] J.C.M. Baeten and J.A. Bergstra. Global renaming operators in concrete process algebra. *Information and Control*, 78:205–245, 1988.

[2] J.C.M. Baeten and J.A. Bergstra. Discrete time process algebra (extended abstract). In W.R. Cleaveland, editor, *CONCUR'92*, pages 401–420. LNCS 630,

Springer-Verlag, 1992. Full version: Report PRG 9208b, Programming Research Group, University of Amsterdam.

[3] J.C.M. Baeten and J.A. Bergstra. Process algebra with signals and conditions. In M. Broy, editor, *Programming and Mathematical Methods*, pages 273–323. NATO ASI Series F88, Springer-Verlag, 1992.

[4] J.C.M. Baeten and J.A. Bergstra. On sequential composition, action prefixes and process prefix. *Formal Aspects of Computing*, 6:250–268, 1994.

[5] J.C.M. Baeten and J.A. Bergstra. Discrete time process algebra. Report P9208c, University of Amsterdam, Programming Research Group, March 1995. To appear in Formal Aspects of Computing.

[6] J.C.M. Baeten and J.A. Bergstra. Discrete time process algebra with abstraction. Report CSR 95-17, Eindhoven University of Technology, Department of Computer Science, 1995. To appear in Proceedings FCT'95.

[7] J.C.M. Baeten and J.A. Bergstra. Some simple calculations in relative time process algebra. Eindhoven University of Technology, Department of Computer Science, 1995. To appear in "Vriendenboek" for Prof. Kruseman Aretz.

[8] J.C.M. Baeten, J.A. Bergstra, and J.W. Klop. Conditional axioms and α/β-calculus in process algebra. In M. Wirsing, editor, *Formal Description of Programming Concepts III*, pages 53–75. North-Holland, 1987.

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

[10] H. Barendregt, H. Wupper, and H. Mulder. Computable processes. Technical Report CSI-R9405, Computing Science Institute, Catholic University of Nijmegen, 1994.

[11] J.A. Bergstra, I. Bethke, and A. Ponse. Process algebra with iteration. *The Computer Journal*, 37:243–258, 1994.

[12] J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Information and Control*, 60:109–137, 1984.

[13] J.A. Bergstra and J.W. Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science*, 37:77–121, 1985.

[14] J.A. Bergstra, J.W. Klop, and E.-R. Olderog. Failures without chaos: A new process semantics for fair abstraction. In M. Wirsing, editor, *Formal Description of Programming Concepts III*, pages 77–103. North-Holland, 1987.

[15] J.A. Bergstra and Gh. Ştefănescu. Network algebra with demonic relation operators. Report P95??, University of Amsterdam, Programming Research Group, 1995.

[16] M.A. Bezem and J.F. Groote. Invariants in process algebra with data. In B. Jonsson and J. Parrow, editors, *CONCUR'94*, pages 401–416. LNCS 836, Springer-Verlag, 1994.

[17] A.P.W. Böhm. *Dataflow Computation.* CWI Tracts 6, Centre for Mathematics and Computer Science, Amsterdam, 1984.

[18] J.D. Brock and W.B. Ackermann. Scenarios: A model of non-determinate computation. In J. Diaz and I. Ramos, editors, *Formalisation of Programming Concepts*, pages 252–259. LNCS 107, Springer-Verlag, 1981.

[19] M. Broy. Nondeterministic dataflow programs: How to avoid the merge anomaly. *Science of Computer Programming*, 10:65–85, 1988.

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

[21] V.E. Căzănescu and Gh. Ştefănescu. A formal representation of flowchart schemes I. *Analele Universitătii Bucuresti, Matematică - Informatică*, 37:33–51, 1988.

[22] V.E. Căzănescu and Gh. Ştefănescu. A formal representation of flowchart schemes II. *Studii si Cercetări Metematice*, 41:151–167, 1989.

[23] V.E. Căzănescu and Gh. Ştefănescu. Towards a new algebraic foundation of flowchart scheme theory. *Fundamenta Informaticae*, 13:171–210, 1990.

[24] I.M. Copy, C.C. Elgot, and J.B. Wright. Realization of events by logical nets. *Journal of the ACM*, 5:181–196, 1958.

[25] G. Dowek, A. Felty, H. Herbelin, G.P. Huet, C. Murthy, C. Parent, C. Paulin-Mohring, and B. Werner. The Coq proof assistant user's guide, version 5.8. Technical report, INRIA – Rocquencourt, May 1993.

[26] C.C. Elgot. Monadic computation and iterative algebraic theories. In H.E. Rose and J.C. Sheperdson, editors, *Logic Colloquium '73*, pages 175–230. Studies in Logic and the Foundations of Mathematics, Volume 80, North-Holland, 1975.

[27] R.J. van Glabbeek. Bounded nondeterminism and the approximation induction principle in process algebra. In F.J. Brandenburg, G. Vidal-Naquet, and M. Wirsing, editors, *STACS 87*, pages 336–347. LNCS 247, Springer-Verlag, 1987.

[28] R.J. van Glabbeek and F.W. Vaandrager. Modular specification of process algebras. *Theoretical Computer Science*, 113:293–348, 1993.

[29] R.J. van Glabbeek and W.P. Weijland. Branching time and abstraction in bisimulation semantics (extended abstract). In G.X. Ritter, editor, *Information Processing 89*, pages 613–618. North-Holland, 1989. Full version: Report CS-9120, CWI.

[30] J.F. Groote and H.P. Korver. A correctness proof of the bakery protocol in μCRL. In A. Ponse, C. Verhoef, and S.F.M. van Vlijmen, editors, *Algebra of Communicating Processes 1994*, pages 63–86. Workshop in Computing Series, Springer-Verlag, 1995.

[31] J.F. Groote and A. Ponse. Proof theory for μCRL: A language for processes with data. In D.J. Andrews, J.F. Groote, and C.A. Middelburg, editors, *Semantics of Specification Languages*, pages 232–251. Workshop in Computing Series, Springer-Verlag, 1994.

[32] J.F. Groote and J. Springintveld. A proof of correctness of the concurrent alternating bit protocol in μCRL. Logic Group Preprint Series ???, Utrecht University, Department of Philosophy, November 1995.

[33] B. Jonsson. A fully abstract trace model for dataflow and asynchronous networks. *Distributed Computing*, 7:197–212, 1994.

[34] G. Kahn. The semantics of a simple language for parallel processing. In J.L. Rosenfeld, editor, *Information Processing '74*, pages 471–475, 1974.

[35] R.M. Keller. Denotational models for parallel programs with nondeterminate operators. In E. Neuhold, editor, *Formal Description of Programming Concepts*, pages 337–366. North-Holland, 1978.

[36] S.C. Kleene. Representation of events in nerve nets and finite automata. In C.E. Shannon and J. McCarthy, editors, *Automata Studies*, pages 3–41. Annals of Mathematical Studies, Volume 34, Princeton University Press, 1956.

[37] J. Kok. A fully abstract semantics for data flow nets. In J.W. de Bakker, A.J. Nijman, and P.C. Treleaven, editors, *PARLE '87*, pages 351–368. LNCS 259, Springer-Verlag, 1987.

[38] H. Korver and J. Springintveld. A computer-checked verification of Milner's scheduler. In M. Hagiya and J. Mitchell, editors, *TACS'94*, pages 161–178. LNCS 789, Springer-Verlag, 1994.

[39] J. Parrow. *Fairness Properties in Process Algebra with Applications in Communication Protocol Verification.* PhD thesis, Department of Computer Science, Uppsala University, 1985.

[40] J. Russell. Full abstraction for nondeterministic dataflow networks. In *FoCS '89*. IEEE Computer Science Press, 1989.

[41] Gh. Ştefănescu. On flowchart theories: Part II. The nondeterministic case. *Theoretical Computer Science*, 52:307–340, 1987.

[42] Gh. Ştefănescu. Feedback theories (a calculus for isomorphism classes of flowchart schemes). *Revue Roumaine de Mathematiques Pures et Applique*, 35:73–79, 1990.

[43] Gh. Ştefănescu. Algebra of flownomials. Part 1: Binary flownomials, basic theory. Technical Report I9437, Department of Computer Science, Technical University Munich, October 1994.

[44] B.C. Thompson and J.V. Tucker. Algebraic specification of synchronous concurrent algorithms and architecture. Technical Report 10-91, Department of Mathematics and Computer Science, University College of Swansea, 1991.

Appendix

We write $[n]$, where $n \in \mathbb{N}$, for $\{1, \ldots, n\}$.

We use the following notation for sequences:

ε	the empty sequence;
x	the sequence having x as sole element;
$\sigma_1 \sigma_2$	the concatenation of the sequences σ_1 and σ_2;
$\|\sigma\|$	the length of the sequence σ;
$hd(\sigma)$	the head of the sequence σ;
$tl(\sigma)$	the tail of the sequence σ;
$\sigma(n)$	the element of the sequence σ with index n;
$\sigma(0..n)$	the prefix of the sequence σ with length $n+1$.

Let $x = (x_1, \ldots, x_m)$ and $y = (y_1, \ldots, y_n)$ be tuples. We write:

$x \frown y$ for $(x_1, \ldots, x_m, y_1, \ldots, y_n)$;

$x(n)$, where x is a tuple of sequences, for $(x_1(n), \ldots, x_m(n))$;

$x(0..n)$, where x is a tuple of sequences, for $(x_1(0..n), \ldots, x_m(0..n))$.

Furthermore, we sometimes use $\langle x, y \rangle$ instead of (x, y).