# Discrete Time Network Algebra for a Semantic Foundation of SDL

J.A. Bergstra, C.A. Middelburg, R. Şoricuţ

October 1997

UNU/IIST Report No. 98

$\boxed{\mathcal{R}}$

# Discrete Time Network Algebra for a Semantic Foundation of SDL

J.A. Bergstra, C.A. Middelburg, R. Şoricuţ

## Abstract

We propose a process algebra model of asynchronous dataflow networks as a semantic foundation for the specification language SDL. The model, which extends a model of network algebra, is close to the concepts around which SDL has been set up. It is able to cover all behavioural aspects of SDL except process creation. More abstract models are derived as well.

Jan Bergstra is a Professor of Programming and Software Engineering at the University of Amsterdam and a Professor of Applied Logic at Utrecht University, both in the Netherlands. His research interest is in mathematical aspects of software and system development, in particular in the design of algebras that can contribute to a better understanding of the relevant issues at a conceptual level. He is perhaps best known for his contributions to the field of process algebra. E-mail: janb@fwi.uva.nl

Kees Middelburg is a Senior Research Fellow at UNU/IIST. He is on a two year leave (1996–1997) from KPN Research and Utrecht University, the Netherlands, where he is a Senior Computer Scientist and a Professor of Applied Logic, respectively. His research interest is in formal techniques for the development of software for reactive and distributed systems, including related subjects such as semantics of specification languages and concurrency theory. E-mail: cam@iist.unu.edu

Radu Şoricuţ is a Fellow at UNU/IIST. He is on a nine month leave (September 1996–May 1997) from Bucharest University, Romania, where he is a last year undergraduate student. His research interest is in concurrency theory and logics for computer science. E-mail: rs@iist.unu.edu

# Contents

**1 Introduction** **1**

**2 Overview of $\varphi$SDL** **2**

**3 Overview of network algebra** **4**
- 3.1 General . . . . . 4
- 3.2 Asynchronous dataflow networks . . . . . 5

**4 Process algebra preliminaries** **6**

**5 Basic network algebra** **8**
- 5.1 Signature and axioms of BNA . . . . . 8
- 5.2 A general process algebra model of BNA . . . . . 9

**6 Asynchronous dataflow networks** **12**
- 6.1 Process algebra model for time-free asynchronous dataflow . . . . . 12
- 6.2 Process algebra model for timed asynchronous dataflow . . . . . 13
- 6.3 Timed asynchronous dataflow networks for SDL . . . . . 15

**7 Dataflow networks for SDL** **17**
- 7.1 Named components . . . . . 17
- 7.2 Composition of named components . . . . . 21
- 7.3 Abstract semantics for SDL networks . . . . . 23

**8 Conclusions** **25**

**A Proofs of main lemmas** **27**

## 1 Introduction

In telecommunications, the language SDL (Specification and Description Language) [24] is widely used for describing structure and behaviour of generally complex telecommunication systems, including switching systems, services and protocols, at different levels of abstraction. The intrinsic highly reactive and distributed nature of the systems developed in telecommunications demands more advanced validation of SDL specifications than currently possible, e.g. validation giving considerations to time dependent behaviour. Besides, the increasing complexity of the systems brings along a growing need to use formal verification to justify design steps. Prerequisites for advanced validation and formal verification is a dramatically simplified version of SDL and an adequate semantics for it. The language $\varphi$SDL [12], has been carefully chosen to meet these requirements.

In [14], network algebra is proposed as a general algebraic setting for the description and analysis of dataflow networks. Such networks represent systems as networks of nodes that consume and produce data and channels between them to pass the data through. Assuming that the components have a fixed number of input and output ports, networks can be built from their components and (possibly branching) connections using parallel composition, sequential composition and feedback. The connections needed are at least the identity and transposition connections, but branching connections may also be needed for specific classes of networks. An equational theory concerning networks that can be built using the above-mentioned operations with only the identity and transposition constants for connections, called BNA (Basic Network Algebra), is presented in [14]. In addition to BNA, an extension for asynchronous dataflow networks is presented. A process algebra model is given as well, thus providing for a very straightforward connection between asynchronous dataflow networks and processes.

In this paper, we adapt that model, that covers the time-free case, to discrete time asynchronous dataflow. We add some atomic components to deal with SDL-like dataflow, viz. mergers, distributors and timers. They are to be used in composing components that correspond to processes in $\varphi$SDL. We also define an operation, corresponding to the kind of composition of processes within a system needed for $\varphi$SDL, in terms of the connections for discrete time asynchronous dataflow and the parallel composition, sequential composition and feedback operations. Thus we obtain a model that is close to the concepts around which SDL has been set up and well suited as the underlying model for a compositional abstract semantics of $\varphi$SDL. Such a semantics is expected to be a suitable starting point for devising proof rules for $\varphi$SDL. From the process algebra model, more abstract models similar to Kahn's history model [19] and Jonsson's trace model [18] are derived. In our opinion, this paper provides convincing mathematical arguments in favour of the choice of concepts concerning storage, communication and timing around which SDL has been set up.

The paper starts with overviews of $\varphi$SDL (Section 2) and network algebra (Section 3), and some process algebra preliminaries (Section 4). A general process algebra model for BNA is presented in Section 5, which is specialised for asynchronous dataflow, timed asynchronous dataflow and further to SDL-like dataflow networks in Section 6. The latter is used in Section 7 in order to define SDL dataflow networks, using a derived operator which acts as a constructor for these networks. More abstract semantics for SDL networks are also proposed.

## 2 Overview of $\varphi$SDL

The language $\varphi$SDL focuses on the behavioural aspects of SDL. The structural aspects of SDL are mostly of a static nature and therefore not very relevant from a semantic point of view. The part of SDL that deals with the specification of abstract data types is well understood. Actually, apart from the data type definitions, SDL system definitions can be transformed to $\varphi$SDL system definitions.

In this section, we give an overview of the concepts around which $\varphi$SDL has been built up. Some peculiar details, inherited from full SDL, are left out to improve the comprehensibility of the overview. These details are, however, made mention of in [12], where a process algebra semantics of $\varphi$SDL is presented.

First of all, the $\varphi$SDL view of a system is explained in broad outline. Basically, a system consists of *processes* which communicate with each other and the environment by sending and receiving *signals* via *signal routes*. A process proceeds in parallel with the other processes in the system and communicates with these processes in an asynchronous manner. This means that a process sending a signal does not wait until the receiving process consumes it, but it proceeds immediately. A process may also use local *variables* for storage of values. A variable is associated with a value that may change by assigning a new value to it. A variable can only be assigned new values by the process to which it is local, but it may be viewed by other processes. Processes can be distinguished by unique addresses, called *pid values* (process identification values), which they get with their creation.

A signal can be sent from the environment to a process, from a process to the environment or from a process to a process. A signal may carry values to be passed from the sender to the receiver; on consumption of the signal, these values are assigned to local variables of the receiver. A signal route is a unidirectional connection between the processes of two types, or between the processes of one type and the environment, for conveying signals. A signal route may contain a *channel*.[1] Signals that must pass through a channel are delayed, but signals always leave a channel in the order in which they have entered it. Thus a signal route is a communication path for sending signals, with or without a delay. If a signal is sent to a process via a signal route that does not contain a channel, it will be instantaneously delivered to that process. Otherwise there may be an arbitrary transmission delay. A channel may be contained in more than one signal route.

A process is either in a *state* or making a *transition* to another state. Besides, when a signal

[1]The original channels from full SDL have been merged with signal routes, but the term channel is reused in $\varphi$SDL.

arrives at a process, it is put into the unique *input queue* associated with the process until it is consumed by the process. The states of a process are the points in its behaviour where a signal may be consumed. However, a state may have signals that have to be saved, i.e. withhold from being consumed in that state. The signal consumed in a state of a process is the first one in its input queue that has not to be saved for that state. If there is no signal to consume, the process waits until there is a signal to consume. So if a process is in a state, it is either waiting to consume a signal or consuming a signal.

A transition from a state of a process is initiated by the consumption of a signal, unless it is a spontaneous transition. A transition is made by performing certain actions: signals may be sent, variables may be assigned new values, new processes may be created and *timers* may be set and reset. A transition may at some stage also take one of a number of branches, but it will eventually come to an end and bring the process to a next state or to its termination.

A timer can be set which sends at its expiration time a signal to the process setting it. A timer is identified with the type and carried values of the signal it sends on expiration. Thus an active timer can be set to a new time or reset; if this is done between the sending of the signal noticing expiration and its consumption, the signal is removed from the input queue concerned. A timer is de-activated when it is reset or the signal it sends on expiration is consumed.

The value of expressions in $\varphi$SDL may vary according to the last values assigned to variables, including local variables of other processes. It may also depend on the system state, e.g. on timers being active or the system time.

In Fig. 1, we give a small example to illustrate how time related behavioural aspects of systems can be specified in $\varphi$SDL. The example concerns the control component of a simple telephone answering machine. The specification is due to Mauw [21]. It is obvious that the behaviour of the control component of an telephone answering machine is time dependent; e.g. the controller should not start the answering immediately when an incoming call is detected.

The simplifications that have been made in $\varphi$SDL with respect to full SDL can be summarised as follows:

- blocks are removed and consequently channels and signal routes are merged;
- variables are treated more liberal: all variables can be viewed freely;
- timer setting is regarded as just a special use of signals;
- timer setting is based on discrete time.

Besides, $\varphi$SDL does not deal with the specification of abstract data types. In this paper, timer setting will be based on relative discrete time.

```
system AnsweringControl
  signal inccall;
  signal endcall;
  signal offhook;
  signal onhook;
  signal beep;
  signal rcvlifted;
  signal playmsg;
  signal endmsg;
  signal startrec;
  signal stoprec;
  signal wtimer;
  signal rtimer;

  signalroute fromnetwork from env to AMC
    with inccall, endcall;
  signalroute tonetwork from AMC to env
    with offhook, onhook, beep;
  signalroute fromtelephone from env to AMC
    with rcvlifted;
  signalroute torecorder from AMC to env
    with playmsg, startrec, stoprec;
  signalroute fromrecorder from env to AMC
    with endmsg;

  process AMC
    start;
      nextstate begin;
    state begin;
      input inccall;
        set(now+10,wtimer);
        nextstate waiting;
    state waiting;
      input endcall;
        reset(wtimer);
        nextstate begin;
      input rcvlifted;
        reset(wtimer);
        nextstate begin;
      input wtimer;
        output offhook via tonetwork;
        output playmsg via torecorder;
        nextstate answering;
    state answering;
      input endcall;
        nextstate end;
      input endmsg;
        output beep via tonetwork;
        output startrec via torecorder;
        set(now+30,rtimer);
        nextstate recording;
    state recording;
      input endcall;
        reset(rtimer);
        output stoprec via torecorder;
        nextstate end;
      input rtimer;
        output stoprec via torecorder;
        nextstate end;
    state end;
      input none;
        output onhook via tonetwork;
        nextstate begin;
  endprocess;
endsystem;
```

Figure 1: An answering machine controller in SDL

## 3 Overview of network algebra

This section gives an idea of what network algebra is. We refer to [13, 14] for extended treatments. The meaning of its operations and constants is explained informally making use of a graphical representation of networks. Besides, asynchronous dataflow networks are presented as a specific class of networks.

### 3.1 General

In the first place, the meaning of the operations and constants of BNA ($+\!\!+$, $\circ$, $\uparrow$, $\mathsf{I}$ and $\mathsf{X}$) is explained. Following, the meaning of additional constants for branching connections is explained.

It is convenient to use, in addition to the operations and constants of BNA, the extensions $\uparrow^m$, $\mathsf{I}_m$ 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 5.1, 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 Fig. 2, 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

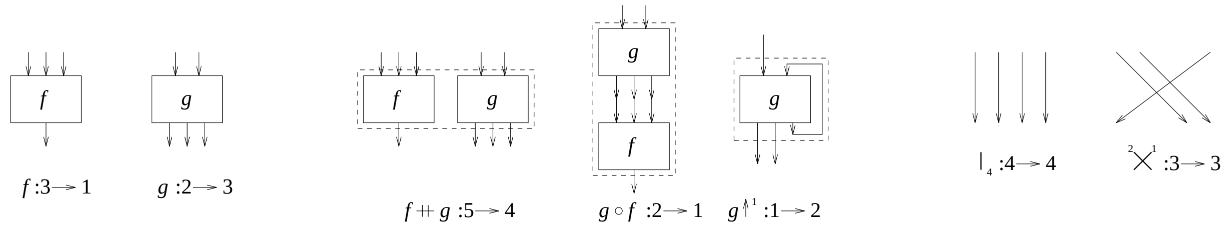

Figure 2: Operations and constants of BNA

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.

## 3.2 Asynchronous dataflow networks

In the case of dataflow networks, the components of a network 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 sequences of data consumed or produced by the cells of a dataflow network are called streams. The wires are interpreted as queues of some kind.

Basic to asynchronous dataflow is that computation is driven by the arrival of the data needed. The underlying idea of asynchronous dataflow is that computation as well as storage and transport of data take a good deal of time. Cells may independently consume data from their input ports, compute new data, and deliver the new data at their output ports. This means that there may be data produced by some cells but not yet consumed by other cells. Therefore the wires have to be able to buffer an arbitrary amount of data.

Dataflow networks also need branching connections. Because there is a flow of data which is everywhere in the network, the interpretation of the branching connections is not immediately clear. Asynchronous dataflow reflects the idea of intermittent flows of data which go in one direction at branchings well. This idea corresponds to the *split/merge* interpretation. We will use the symbols $\wedge$ and $\vee$ for split and merge, respectively. Dataflow networks have been extensively studied, see e.g. [15, 16, 18, 19, 20, 22].

## 4 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 the following sections. 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 [10], extended with the silent step $\tau$ and the abstraction operator $\tau_I$ for abstraction. Semantically, we adopt the approach to abstraction, originally proposed for ACP in [17], which is based on branching bisimulation. ACP with this kind of abstraction is called ACP$^\tau$. 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 $(P_i)_{i \in J}$ be an indexed set of process expressions where $J = \{i_1, \ldots, i_n\}$. Then, we write $\sum_{i \in J} P_i$ for $P_{i_1} + \ldots + P_{i_n}$, and $\|_{i \in J} P_i$ for $P_{i_1} \parallel \ldots \parallel P_{i_n}$. We further use the convention that $\sum_{i \in J} P_i$ and $\|_{i \in J} P_i$ stand for $\delta$ if $J = \emptyset$. For a systematic introduction to ACP, the reader is referred to [8].

Further we will use the following extensions:

**renaming** We need the possibility of renaming actions. We will use the renaming operator $\rho_f$, added to ACP in [1]. Here $f$ is a function that renames actions into actions, $\delta$ or $\tau$. 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 $\lhd \bullet \rhd$ as in [3]. The expression $P \lhd b \rhd Q$, is to be read as $\mathsf{if}\, b\, \mathsf{then}\, P\, \mathsf{else}\, Q$. 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]. It 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 [9], 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:

$$\begin{array}{rcl}(er_1(x_1) \parallel er_2(x_2))\ ;\ P & = & \displaystyle\sum_{d_1 \in D} r_1(d_1) \cdot (er_2(x_2)\ ;\ P[d_1/x_1]) \\ & & + \displaystyle\sum_{d_2 \in D} r_2(d_2) \cdot (er_1(x_1)\ ;\ P[d_2/x_2])\end{array}$$

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

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 $P \mathbin{|||} Q = P \mathbin{|||\!\lfloor} Q + Q \mathbin{|||\!\lfloor} P$, where $|||\!\lfloor$ is defined as $|\!\lfloor$ except that $a \cdot P \mathbin{|||\!\lfloor} Q = a \cdot (P \mathbin{|||} Q)$. Communication free merge can also be expressed in terms of parallel composition, encapsulation and renaming.

**priority** The priority operator $\theta$ was originally introduced in [7]. It uses a partial order on the atomic actions which is used to choose from the actions with the highest priority in alternative composition. In order to describe it, an auxiliary operator $\lhd$ (unless) is needed. The crucial equation is $\theta(x+y) = \theta(x) \lhd y + \theta(y) \lhd x$. Here $\lhd$ behaves like a filter: $a \lhd b = a$ unless $a < b$ holds in the partial ordering; in that case $a \lhd b = \delta$.

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

Time is divided into slices representing 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 $\sigma_{\mathsf{rel}}$. 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 [6], 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 [5], the notation $\mathsf{cts}(\tau)$ is used instead of $\underline{\underline{\tau}}$. The process $a$ is defined in terms $\underline{\underline{a}}$ and $\sigma_{\mathsf{rel}}$ 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 $P_i$s. 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 ACP$_{\mathrm{drt}}$. The integration of renaming, iteration, communication free merge and priority 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 $\underline{\underline{er}}_i(x)\ ;\ P = \sum_{d \in D} \underline{\underline{r}}_i(d) \cdot P[d/x]$ and $\sigma_{\mathsf{rel}}(P)\ ;\ Q = \sigma_{\mathsf{rel}}(P\ ;\ Q)$.

**conditionals and discrete time** The discrete time extension requires a new conditional operator. This is the two-armed sliced conditional operator $\lhd \bullet \rhd$. The expression $P \lhd b \rhd Q$ can be expressed using a one-armed sliced conditional operator: $P \lhd b \rhd Q = b \mathrel{\vdots\!\!\rightarrow} P + \neg b \mathrel{\vdots\!\!\rightarrow} Q$, where $(\mathsf{t} \mathrel{\vdots\!\!\rightarrow} P) = P$ and $(\mathsf{f} \mathrel{\vdots\!\!\rightarrow} P) = \underline{\underline{\delta}}$. The new operator requires an additional axiom (due to Yaroslav Usenko): $\sigma_{\mathsf{rel}}(P) \mathbin{|\!\lfloor} (Q \lhd \phi \rhd R) = (\sigma_{\mathsf{rel}}(P) \mathbin{|\!\lfloor} Q) \lhd \phi \rhd (\sigma_{\mathsf{rel}}(P) \mathbin{|\!\lfloor} R)$.

## 5 Basic network algebra

In this section, BNA is presented. First of all, the signature and axioms of BNA are given. In addition, a general process algebra model of BNA is described. In a subsequent section, extensions of BNA for asynchronous dataflow networks are provided.

### 5.1 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 | $+\!\!+$ | $(k \to l) \times (m \to n) \to (k + m \to l + n)$ |
| 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 | $\mathsf{I}$ | $1 \to 1$ |
| transposition | $\mathsf{X}$ | $2 \to 2$ |

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 3, 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 |
|---|---|
| $\uparrow^{l}$ | $(m+l \to n+l) \to (m \to n)$ |
| $\mathsf{I}_m$ | $m \to m$ |
| ${}^{m}\mathsf{X}^{n}$ | $m+n \to n+m$ |

### Axioms

The axioms of BNA are given in Table 1. The axioms B1–B10 are concerned with $\mathbin{+\!\!+}$, $\circ$, $\mathsf{I}_m$

| | | | |
|---|---|---|---|
| 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

and ${}^{m}\mathsf{X}^{n}$ and the remaining axioms characterise $\uparrow^{l}$. The axioms R5–R6, B6 and B8–B9 can be regarded as the defining equations of the block extensions of $\uparrow$, $\mathsf{I}$ and $\mathsf{X}$, respectively. The axioms of BNA are sound and complete for basic networks modulo graph isomorphism (cf. [23]).

As a first step towards the process algebra models for asynchronous dataflow networks described in Section 6, a general process algebra model of BNA is provided.

## 5.2 A general process algebra model of BNA

Network algebra can be regarded as being built on top of process algebra. A process algebra model of BNA is presented in [14], and this model is specialised to give a model for asynchronous dataflow. Here we follow a similar approach, but we allow to give priorities to certain atomic actions. The definitions for sequential composition and feedback are modified, using the priority operator $\theta$. It should be noticed that our new definitions do not change sequential composition and feedback in an essential way: in case the atomic actions are unordered (i.e., for all actions $a$ and $b$, $a \not< b$), our definitions are equivalent to the ones from [14]. This means that the modified definitions turn out to be more general.

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

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, the 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 $\delta$.

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

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

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

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

(P3) $\tau_{I(u,v)}(\theta(\partial_{H(u,v)}((\rho_{out(j/v)}(P) \mathbin{|\!|\!|} w_j^u) \parallel w_u^v))) = P$ 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 $\mathsf{GProc}(D)$ as follows:

| Name | Notation | | |
|---|---|---|---|
| parallel composition | $f \mathbin{+\!\!\!+} g$ | $\in \mathsf{GProc}(D)(m+p, n+q)$ | for $f \in \mathsf{GProc}(D)(m,n)$, $g \in \mathsf{GProc}(D)(p,q)$ |
| seq. composition | $f \circ g$ | $\in \mathsf{GProc}(D)(m,p)$ | for $f \in \mathsf{GProc}(D)(m,n)$, $g \in \mathsf{GProc}(D)(n,p)$ |
| feedback | $f \uparrow^p$ | $\in \mathsf{GProc}(D)(m,n)$ | for $f \in \mathsf{GProc}(D)(m+p, n+p)$ |
| identity | $\mathsf{I}_n$ | $\in \mathsf{GProc}(D)(n,n)$ | |
| transposition | ${}^m\mathsf{X}^n$ | $\in \mathsf{GProc}(D)(m+n, n+m)$ | |

Definition

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

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

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

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

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

□

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 $\delta$, $\tau \cdot \delta$ 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 $\mathrm{ACP}^{\tau}$ and $\mathrm{ACP}^{\tau}_{\mathrm{drt}}$ or belong to a few of their mutual (conservative) extensions mentioned in Section 4 (viz. renaming, communication free merge and priority). As a result, we can specialise this general model for a specific kind of networks using either $\mathrm{ACP}^{\tau}$ or $\mathrm{ACP}^{\tau}_{\mathrm{drt}}$ with further extensions at need.

**Theorem 5.2** $(\mathsf{GProc}(D), \mathbin{+\!\!\!+}, \circ, \uparrow, \mathsf{I}, \mathsf{X})$ *is a model of BNA if actions are not ordered.*

**Proof:** When atomic actions are not ordered, this result reduces to Theorem 4.4 in [14]. □

Later it will be shown that this result also holds for the non-trivial order on atomic actions introduced in Section 6.3.

So if we select a specific wire, as we do in Section 6, we obtain 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 $f = f \circ \mathsf{I}_n$.

## 6 Asynchronous dataflow networks

In this section, specialisations of the process algebra model of Section 5.2 for asynchronous dataflow networks are described. We present models for time-free and timed asynchronous dataflow networks, as well as a model of timed asynchronous dataflow for SDL.

### 6.1 Process algebra model for time-free asynchronous dataflow

In this subsection, the specialisation of the process algebra model of BNA (Section 5.2) for time-free asynchronous dataflow networks is given. In this case, we will make use of ACP$^\tau$.

In Section 5.2, only a few assumption about wires and atomic cells were made. In this subsection these ingredients are actualized for asynchronous dataflow networks in the time-free case.

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

$$\mathsf{sd}_1^1(\sigma) = er_1(x) \mathbin{;} \mathsf{sd}_1^1(\sigma \frown x) + |\sigma| > 0 :\!\rightarrow s_1(hd(\sigma)) \cdot \mathsf{sd}_1^1(tl(\sigma))$$

An atomic cell with $m$ inputs and $n$ outputs is a network

$$C = \mathsf{I}_m \circ (m, n, P) \circ \mathsf{I}_n$$

where $P$ is a process with actions in $\{r_i(d) \mid i \in [m], d \in D\} \cup \{s_i(d) \mid i \in [n], d \in D\}$.

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

The definition of $\mathsf{sd}_1^1$ simply expresses that it behaves as a queue. The definition of atomic cells shows that the buffering it needs because of the asynchronous dataflow is built in.

For $\mathsf{AProc}(D)$, the operations and constants of BNA as defined on $\mathsf{GProc}(D)$ 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 untimed asynchronous dataflow)
The operations $+\!\!+$, $\circ$, $\uparrow^n$ on $\mathsf{AProc}(D)$ are the instances of the ones defined on $\mathsf{GProc}(D)$ for $\mathtt{sd}^1_1$ as wire. Analogously, the constants $\mathsf{I}_n$ and ${}^m\mathsf{X}^n$ in $\mathsf{AProc}(D)$ are the instances of the ones defined on $\mathsf{GProc}(D)$ for $\mathtt{sd}^1_1$ as wire.

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

| Name | Notation | |
|---|---|---|
| split | ⋏ | $\in \mathsf{AProc}(D)(1,2)$ |
| sink | ᑲ | $\in \mathsf{AProc}(D)(1,0)$ |
| merge | ⋎ | $\in \mathsf{AProc}(D)(2,1)$ |
| dummy source | ⫯ | $\in \mathsf{AProc}(D)(0,1)$ |

| Definition | | | |
|---|---|---|---|
| ⋏ | $=$ | $\mathsf{I}_1 \circ (1, 2, split^1) \circ \mathsf{I}_2$ | where $split^1 = (er_1(x) \;;\; (s_1(x) + s_2(x)))\ ^*\ \delta$ |
| ᑲ | $=$ | $\mathsf{I}_1 \circ (1, 0, sink^1)$ | where $sink^1 = (er_1(x) \;;\; \tau)\ ^*\ \delta$ |
| ⋎ | $=$ | $\mathsf{I}_2 \circ (2, 1, merge_1) \circ \mathsf{I}_1$ | where $merge_1 = ((er_1(x) + er_2(x)) \;;\; s_1(x))\ ^*\ \delta$ |
| ⫯ | $=$ | $(0, 1, source_1) \circ \mathsf{I}_1$ | where $source_1 = \delta$ |

□

**Theorem 6.3** $(\mathsf{AProc}(D), +\!\!+, \circ, \uparrow, \mathsf{I}, \mathsf{X})$ *is a model of BNA if actions are not ordered.*

**Proof:** When the atomic actions are not ordered, this result reduces to the first part of Theorem 5.4 in [14]. □

## 6.2 Process algebra model for timed asynchronous dataflow

In this subsection, the specialisation of the process algebra model of BNA (Section 5.2) for timed asynchronous dataflow networks is given. In this case, we will make use of $\mathrm{ACP}^{\tau}_{\mathrm{drt}}$.

First wires and atomic cells are actualised for timed asynchronous dataflow networks. This is similar to the actualisation for time-free asynchronous dataflow networks given in Section 6.1.

**Definition 6.4** (wires and atomic cells in timed asynchronous dataflow networks)
In the timed asynchronous case, the identity constant, now called the *timed stream delayer*, is the wire $\mathsf{I}_1 = (1, 1, \mathtt{tsd}_1^1(\varepsilon))$, where $\mathtt{tsd}_1^1$ is defined by

$$\mathtt{tsd}_1^1(\sigma) = er_1(x) \ ; \mathtt{tsd}_1^1(x) \lhd |\sigma| = 0 \rhd (\underline{\underline{er}}_1(x) \ ; \mathtt{tsd}_1^1(\sigma ^\frown x) + \underline{\underline{s}}_1(hd(\sigma)) \cdot \mathtt{tsd}_1^1(tl(\sigma)))$$

An atomic cell with $m$ inputs and $n$ outputs is a network

$$C = \mathsf{I}_m \circ (m, n, P) \circ \mathsf{I}_n$$

where $P$ is a process with actions in $\{r_i(d) \mid i \in [m], d \in D\} \cup \{s_i(d) \mid i \in [n], d \in D\}$. The restriction of $\mathsf{GProc}(D)$ to the processes that can be built under this actualisation is denoted by $\mathsf{TAProc}(D)$. $\square$

The definition of $\mathtt{tsd}_1^1$ expresses that it behaves as a queue, it is able to contain an arbitrary amount of data, but data will always enter and leave it within the same time slice. The definition of atomic cells is the same as in the time-free case.

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

**Definition 6.5** (process algebra model for timed asynchronous dataflow)
The operations $+\!\!+$, $\circ$, $\uparrow^n$ and the constants $\mathsf{I}_n$ and ${}^m\mathsf{X}^n$ in $\mathsf{TAProc}(D)$ are the instances of the ones defined on $\mathsf{GProc}(D)$ for $\mathtt{tsd}_1^1$ as wire.

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

| Name | Notation | |
|---|---|---|
| split | $\overset{\bullet}{\wedge}$ | $\in \mathsf{TAProc}(D)(1,2)$ |
| sink | $\overset{|}{\circ}$ | $\in \mathsf{TAProc}(D)(1,0)$ |
| merge | $\underset{\bullet}{\vee}$ | $\in \mathsf{TAProc}(D)(2,1)$ |
| dummy source | $\overset{\bullet}{|}$ | $\in \mathsf{TAProc}(D)(0,1)$ |

Definition

$$\begin{array}{lcll}
⋏ & = & \mathsf{I}_1 \circ (1,2,\mathit{split}^1) \circ \mathsf{I}_2 & \text{where } \mathit{split}^1 = (er_1(x)\ ;\ (\underline{\underline{s}}_1(x) + \underline{\underline{s}}_2(x)))\ ^*\ \underline{\underline{\delta}} \\
⫰ & = & \mathsf{I}_1 \circ (1,0,\mathit{sink}^1) & \text{where } \mathit{sink}^1 = (er_1(x)\ ;\ \underline{\underline{\tau}})\ ^*\ \underline{\underline{\delta}} \\
⋎ & = & \mathsf{I}_2 \circ (2,1,\mathit{merge}_1) \circ \mathsf{I}_1 & \text{where } \mathit{merge}_1 = ((er_1(x) + er_2(x))\ ;\ \underline{\underline{s}}_1(x))\ ^*\ \underline{\underline{\delta}} \\
⫯ & = & (0,1,\mathit{source}_1) \circ \mathsf{I}_1 & \text{where } \mathit{source}_1 = \delta
\end{array}$$

□

**Lemma 6.6** *For the wire* $\mathsf{I}_1 = (1,1,\mathtt{tsd}_1^1(\varepsilon))$, $\mathsf{I}_1 \circ \mathsf{I}_1 = \mathsf{I}_1$.

**Proof:** The proof of this lemma is given in the appendix. □

**Lemma 6.7** *The wire* $\mathsf{I}_1 = (1,1,\mathtt{tsd}_1^1)$ *gives an identity flow of data, i.e. for all* $f = (m,n,P)$ *in* $\mathsf{TAProc}(D)$, $\mathsf{I}_m \circ f = f = f \circ \mathsf{I}_n$.

**Proof:** By Lemma 6.6 we have that $\mathsf{I}_1 \circ \mathsf{I}_1 = \mathsf{I}_1$. $\mathsf{I}_n \circ \mathsf{I}_n = \mathsf{I}_n$ 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 $\mathsf{I}_n$ and ${}^m\mathsf{X}^n$. Due to the pre- and postfixing with identities in the definitions of the remaining constants and the cells, it follows trivially that these equations hold also for them. The result then follows by induction on the construction of a network in $\mathsf{TAProc}(D)$. □

**Theorem 6.8** $(\mathsf{TAProc}(D), \mathbin{+\!\!+}, \circ, \uparrow, \mathsf{I}, \mathsf{X})$ *is a model of BNA if actions are not ordered.*

**Proof:** A simple calculation shows that $\mathsf{I}_0 \mathbin{+\!\!+} f = f = f \mathbin{+\!\!+} \mathsf{I}_0$ for all $f \in \mathsf{TAProc}(D)$. Then the theorem follows immediately from Theorem 5.2 and Lemma 6.7. □

## 6.3 Timed asynchronous dataflow networks for SDL

In this subsection we give another process algebra model for timed asynchronous dataflow which will be mainly based on the definitions of $\mathsf{TAProc}(D)$. The difference is that a non-trivial partial order on atomic actions is given such that the new model can deal with asynchronous dataflow networks corresponding to systems described in SDL.

Let $\mathcal{C}$ be a fixed, but arbitrary, set of process names. $\mathcal{C}$ is an additional parameter of the model. We write $\mathcal{D}$ for the cartesian product

$$(\mathcal{C} \cup \{\mathbf{env}\} \cup \{\mathsf{timer}\} \cup \{\mathsf{setr}(i) \mid i \in \mathbb{N}\} \cup \{\mathsf{reset}\} \cup \{\mathsf{nil}\}) \times D$$

where $\mathsf{timer}, \mathsf{setr}(i), \mathsf{reset} \notin \mathcal{C}$ $(i \in \mathbb{N})$. The use of $\mathsf{timer}, \mathsf{setr}(i), \mathsf{reset}$ and $\mathcal{C}$ will be explained in Section 7.1. The processes now use the standard actions $r_i(d)$, $s_i(d)$ and $c_i(d)$ for $d \in \mathcal{D}$. We define the priority relation $<$ as the least partial order relation such that

$$x < c_i((\mathsf{reset}, y)) \text{ and } x < s_i((\mathsf{reset}, y))$$

for all actions $x \notin \{c_i((\mathsf{reset}, y)) | \ i \in \mathbb{N},\ y \in D\} \cup \{s_i((\mathsf{reset}, y)) | \ i \in \mathbb{N},\ y \in D\}$.

**Definition 6.9** (wires and atomic cells for dataflow networks with SDL-timers)
The identity constant is the wire $\mathsf{I}_1 = (1, 1, \mathtt{ssd}_1^1(\varepsilon))$, where $\mathtt{ssd}_1^1$ is defined by

$$\begin{array}{l}
\mathtt{ssd}_1^1(\sigma) = \\
(er_1((x,y)) \ ; \ (\underline{\underline{s}}_1((\mathsf{reset}, y)) \cdot \mathtt{ssd}_1^1(\varepsilon) \lhd x = \mathsf{reset} \rhd \mathtt{ssd}_1^1((x,y)))) \lhd |\sigma| = 0 \rhd \\
(\underline{\underline{er}}_1((x,y)) \ ; \ (\underline{\underline{s}}_1((\mathsf{reset}, y)) \cdot \mathtt{ssd}_1^1(reset(\sigma, y)) \lhd x = \mathsf{reset} \rhd \mathtt{ssd}_1^1(\sigma^\frown(x,y))) + \\
\underline{\underline{s}}_1(hd(\sigma)) \cdot \mathtt{ssd}_1^1(tl(\sigma)))
\end{array}$$

where $reset(\sigma, d)$, $d \in D$, stands for the sequence $\sigma$ with all the occurrences of the data $(\mathsf{timer}, d)$ and $(\mathsf{setr}(i), d)$, for any $i \in \mathbb{N}$, removed from it.

The atomic cells are defined as in Definition 6.4.

The restriction of $\mathsf{GProc}(D)$ to the processes that can be built under this actualisation is denoted by $\mathsf{SDLProc}(\mathcal{C}, D)$. $\square$

The definition of $\mathtt{ssd}_1^1$ expresses that it normally behaves as a queue, it is able to contain an arbitrary amount of data, but the data will always enter and leave it within the same time slice. However, if a datum $(\mathsf{reset}, y)$ enters it, all data $(\mathsf{timer}, y)$ and $(\mathsf{setr}(i), y)$ are removed, and $(\mathsf{reset}, y)$ leaves it before any other datum has entered of left.

**Definition 6.10** (process algebra model for dataflow networks with SDL-timers)
The operations $+\!\!+$, $\circ$, $\uparrow^n$ and the constants $\mathsf{I}_n$ and ${}^m\mathsf{X}^n$ in $\mathsf{SDLProc}(\mathcal{C}, D)$ are the instances of the ones defined on $\mathsf{GProc}(D)$ for $\mathtt{ssd}_1^1$ as wire.

The additional constants in $\mathsf{SDLProc}(\mathcal{C}, D)$ are defined as in Definition 6.5 $\square$

**Lemma 6.11** *For the wire* $\mathsf{I}_1 = (1, 1, \mathtt{ssd}_1^1(\varepsilon))$, $\mathsf{I}_1 \circ \mathsf{I}_1 = \mathsf{I}_1$.

**Proof:** The proof of this lemma is given in the appendix. $\square$

**Theorem 6.12** $(\mathsf{SDLProc}(\mathcal{C}, D), \mathbin{+\!\!+}, \circ, \uparrow, \mathsf{I}, \mathsf{X})$ *is a model of BNA if the priority relation given above is used.*

**Proof:** By Lemma 6.11 the proof of Theorem 6.8 carries over to this theorem. □

## 7 Dataflow networks for SDL

In this section we make additions to the process algebra model $\mathsf{SDLProc}(\mathcal{C}, D)$ from Section 6.3 to obtain a model of networks representing SDL systems. We define some atomic components that are to be used in composing components that correspond to processes in SDL. We also explain how SDL processes fit into our framework. And an operation is defined, in terms of the connections for discrete time asynchronous dataflow and the parallel composition, sequential composition and feedback operations, corresponding to the kind of composition of processes within a system needed for SDL.

### 7.1 Named components

For each name in $\mathcal{C}$ there is a corresponding named component. A named component is built from a merger, a distributor, a timer, and a main cell. The main cell makes use of the following read and send actions only:

| | |
|---|---|
| $r_1((c, d))$ | reading datum $d$ from $c \in \mathcal{C} \cup \{\mathbf{env}\}$ |
| $r_1((\mathsf{timer}, d))$ | reading the expiration notification of timer $d$ |
| $r_1((\mathsf{reset}, d))$ | reading the reset notification of timer $d$ |
| $s_1((c, d))$ | sending datum $d$ to $c \in \mathcal{C} \cup \{\mathbf{env}\}$ |
| $s_2((\mathsf{setr}(i), d))$ | setting the timer $d$ to $i$ units from now |
| $s_2((\mathsf{reset}, d))$ | resetting the timer $d$ |

A main cell has only one input port and two output ports; input port 1 and output port 1 are meant for communication with other named components and the environment, while output port 2 is meant for setting and resetting of timers.

It is further assumed that there is a bijection $p : [|\mathcal{C}|+1] \to \mathcal{C} \cup \{\mathbf{env}\}$ such that $p(|\mathcal{C}|+1) = \mathbf{env}$. The bijection reflects the way the named components are connected, namely such that at the input port $i$ data from component $p(i)$ is consumed and at the output port $i$ data for component $p(i)$ is produced. This explain how mergers and distributors transform data. When a pair $(\mathsf{nil}, d)$ is offered at input port $i$, a merger produces the pair $(p(i), d)$ at its only output port. When a pair $(p(i), d)$ is offered at its only input port, a distributor produces the pair $(\mathsf{nil}, d)$ at output port $i$.

In the definition of a timer cell below, a process $\mathtt{timer}(\alpha)$ is defined for each infinite sequence $\alpha$ of finite sets of data. The process $\mathtt{timer}(\alpha)$ is informed that $i$ time units from now the timers in the set $\alpha(i)$ expire (for $i \in \mathbb{N}$).

**Definition 7.1** (additional atomic cells for SDL)
A $n$-merger is a cell $\mathtt{MERGER}_n = \mathsf{I}_n \circ (n, 1, \mathtt{merger}_n) \circ \mathsf{I}_1$, where $\mathtt{merger}_n$ is defined by

$$\mathtt{merger}_n = (\sum_{i \in [n]} er_i((\mathsf{nil}, x)) \; ; \underline{\underline{s}}_1((p(i), x))) * \underline{\underline{\delta}}$$

Similarly, a $n$-distributor is a cell $\mathtt{DISTRIBUTOR}_n = \mathsf{I}_1 \circ (1, n, \mathtt{distributor}_n) \circ \mathsf{I}_n$, where $\mathtt{distributor}_n$ is defined by

$$\mathtt{distributor}_n = (\sum_{i \in [n]} er_1((p(i), x)) \; ; \underline{\underline{s}}_i((\mathsf{nil}, x))) * \underline{\underline{\delta}}$$

A timer is a cell $\mathtt{TIMER} = \mathsf{I}_1 \circ (1, 1, \mathtt{timer}(\emptyset^\frown \emptyset^\frown \ldots)) \circ \mathsf{I}_1$, where

$$\begin{aligned}
\mathtt{timer}(\alpha) = \;& \mathtt{timer}'(tl(\alpha)) \lhd hd(\alpha) = \emptyset \rhd \\
& ( \;\|_{d \in hd(\alpha)} \; \underline{\underline{s}}_1((\mathsf{timer}, d)) \cdot \mathtt{timer}'(tl(\alpha))) \\[1em]
\mathtt{timer}'(\alpha) = \;& er_1((\mathsf{setr}(i), x)) \; ; \mathtt{timer}'(upd(\alpha, i, x)) + \\
& er_1((\mathsf{reset}, x)) \; ; \underline{\underline{s}}_1((\mathsf{reset}, x)) \cdot \mathtt{timer}'(rem(\alpha, x)) + \\
& \textstyle\sum_{c \in \mathcal{C} \cup \{\mathsf{env}\}} er_1((c, x)) \; ; \underline{\underline{s}}_1((c, x)) \cdot \mathtt{timer}'(\alpha) + \\
& \sigma_{\mathsf{rel}}(\mathtt{timer}(\alpha))
\end{aligned}$$

where we write $upd(\alpha, i, d)$ for the infinite sequence $\alpha'$ such that $\alpha'(i) = \alpha(i) \cup \{d\}$ and $\alpha'(j) = \alpha(j) - \{d\}$ for all $j \in \mathbb{N}, j \neq i$; and $rem(\alpha, d)$ for the infinite sequence $\alpha'$ such that $\alpha'(j) = \alpha(j) - \{d\}$ for all $j \in \mathbb{N}$.

□

The definition of $\mathtt{timer}$ expresses that there are two phases in the behaviour of timers during a time slice. In one phase, for each timer that expires in the current time slice, a datum representing expiration notification is produced at its only output port, and it does so in arbitrary order. The expiration notification data are of the form $(\mathsf{timer}, d)$. In the other stage, it consumes data representing timer setting and resetting requests. The purpose of sending $(\mathsf{reset}, d)$ is to cover the following aspect of the SDL-timer mechanism: if a datum representing expiration notification has been produced but not yet consumed and the timer concerned is set again or reset, this datum has to be removed. The non-trivial priority relation and the wire $\mathtt{ssd}_1^1$ are needed to do so instantaneously. Besides, in this phase it consumes and delivers data received from other processes and from the environment.

**Definition 7.2** (named component)
Let $n$ be the number of names in $\mathcal{C}$. To each name $c \in \mathcal{C}$ we will assign a network $N_c$ of sort $n+1 \to n+1$, called a named component, where

$$N_c = \mathtt{MERGER}_{n+1} \circ ((\curlyvee \circ \mathtt{TIMER} \circ C_c) \uparrow^1) \circ \mathtt{DISTRIBUTOR}_{n+1}$$

for some main cell $C_c$ of sort $1 \to 2$. □

The graphical representation of a named component is given in Figure 3. Named components

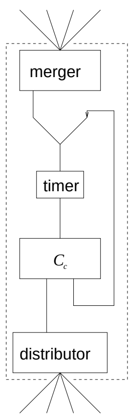

Figure 3: The named component $c$

correspond to processes in SDL.

## Constructing a main cell

The main cells $C_c$, for $c \in \mathcal{C}$, are parameters for our construction, but they are meant to correspond to SDL processes. We describe how to model, for some states of an SDL process, the behaviour of the corresponding main cell by means of recursive specifications in $\mathrm{ACP}^{\tau}_{\mathrm{drt}}$-ID.

Assume that we have the following signals and signal routes in a given SDL description:

```
signal Sig;                      signalroute from_env from env to c5 with Sig2;
signal Sig';                     signalroute to_env from c5 to env with Sig,Sig3;
signal Sig1;                     signalroute from_c1 from c1 to c5 with Sig1;
signal Sig2;                     signalroute to_c2 from c5 to c2 with Sig2;
signal Sig3;                     signalroute from_c3 from c3 to c5 with Sig3;
signal Sig4;                     signalroute from_c4 from c4 to c5 with Sig4;
```

We take $\mathcal{C}$ and $D$ such that $c1, \ldots, c5 \in \mathcal{C}$ and $Sig, Sig', Sig1, \ldots, Sig4 \in D$. The SDL processes with names $c1, \ldots, c5$ correspond to the main cells of named components with these names.

Each SDL process is either in a state or making a transition. We describe how to model, for some states of the SDL process $c5$, the behaviour of the corresponding main cell by means of recursive specifications in $\mathrm{ACP}^{\tau}_{\mathrm{drt}}$-ID.

We use the notation $Disc(S)$ for set of discarded signals in state $S$, i.e. the set $\mathcal{D}$ without what is explicitly expected as signals in that state.

The input queue of the process is the sequence of data from the incoming wire of the main cell, i.e. the wire $\mathsf{I}_1$ in the construction $C_c = \mathsf{I}_1 \circ (1, 2, P) \circ \mathsf{I}_2$. The consumption of a signal by the process $P$ is the communication action between the process $\mathtt{ssd}^1_1$ that makes up its input queue and the process $P$. The behaviour of the main cell corresponding to the SDL process $c5$ from the states that are presented in Figure 4 – using the graphical representation form of SDL – is described by the following equations:

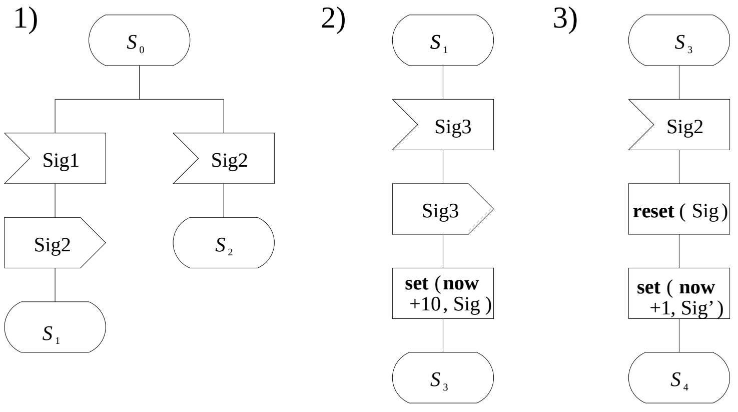

Figure 4: SDL states and transitions

$$
\begin{aligned}
S_0 &= r_1((c1, Sig1)) \cdot \underline{\underline{s}}_1((c2, Sig2)) \cdot S_1 + r_1((\mathbf{env}, Sig2)) \cdot S_2 + \\
&\quad \sum_{(x,y)\in Disc(S_0)} r_1((x, y)) \cdot S_0 \\
S_1 &= r_1((c3, Sig3)) \cdot \underline{\underline{s}}_1((\mathbf{env}, Sig3)) \cdot \underline{\underline{s}}_2((\mathsf{reset}, Sig)) \cdot \underline{\underline{s}}_2((\mathsf{setr}(10), Sig)) \cdot S_3 + \\
&\quad \sum_{(x,y)\in Disc(S_1)} r_1((x, y)) \cdot S_1 \\
S_3 &= r_1((\mathbf{env}, Sig2)) \cdot \underline{\underline{s}}_2((\mathsf{reset}, Sig)) \cdot \underline{\underline{s}}_2((\mathsf{reset}, Sig')) \cdot \underline{\underline{s}}_2((\mathsf{setr}(1), Sig')) \cdot S_4 + \\
&\quad \sum_{(x,y)\in Disc(S_3)} r_1((x, y)) \cdot S_3
\end{aligned}
$$

Setting a timer must be preceded by a reset request of the same timer. Note that a timer can be set using relative time only, i.e. $\underline{\underline{s}}_2((\mathsf{setr}(i), Sig))$ is a request for setting the timer referred to by $Sig$ for $i$ time units from now.

According to [24], SDL processes may not send signals to themselves. The same restriction applies the processes that make up the main cells. That is, the process that make up the main cell $C_c$ should not perform actions like $r_1((c, sig))$ or $s_1((c, sig))$, for any $sig \in D$. This restriction can be built-in. Recall that $p$ is the bijection reflecting the way the named components are connected. For each named component $N_{p(i)}$ ($i \in [n]$), the input and output port number $i$ can be eliminated, and the remaining ports can be renamed from 1 to $n$. However, in that case we would need a different merger and distributor for each $i \in [n]$.

Due to the special treatment the wire $\mathtt{ssd}_1^1$ offers to data of the form $(\mathsf{reset}, y)$, it seems that we do not model dataflow networks: the first-in-first-out discipline is not respected by our wires. However, note that we only have data of the form $(\mathsf{reset}, y)$ inside a named component. If we regard the named components as black-boxes, we only see wires that behaves as the wires $\mathtt{tsd}_1^1$.

## 7.2 Composition of named components

In this subsection, we define networks representing SDL systems, which we will call SDL networks. First we define an operation, called the inter-connection operation, to compose an SDL network from named components. The notion of an SDL context will be introduced as well. First of all, some auxiliary networks are introduced.

In order to build up an SDL network from named components, we need to make connections between them. The network $\mathtt{F}_n$ will make these connections.

**Definition 7.3** (connections between named components)
Let $f_n : [n^2] \to [n^2]$, for every natural number $n \geq 1$, be the bijection:

$$f_n(i) = n((i-1) \bmod n) + (i-1) \div n + 1$$

where $\div$ and mod are integer division and modulo, respectively.

We define the network $\mathtt{F}_n$ representing the bijection $f_n$ as follows:

$$\mathtt{F}_n = \mathsf{I}_{n^2} \circ (n^2, n^2, \mathtt{f_n}) \circ \mathsf{I}_{n^2}$$

where $\mathtt{f}_n$ is defined by

$$\mathtt{f}_n = (\sum_{i \in [n^2]} er_i(x) \; ; \underline{\underline{s}}_{f_n(i)}(x)) \, {}^* \, \underline{\underline{\delta}}$$

□

Cf. [23], any bijection can be represented by a network, using identities, transpositions and parallel and sequential composition, only.

An SDL network containing $n$ named components is of sort $n \to n$, which means that it has $n$ input ports and $n$ output ports. The network $\mathtt{ITF}_n$ is used to connect the input port $i$ of the SDL network to the input port $n+1$ of the named component $N_{p(i)}$, for each $i \in [n]$.

**Definition 7.4** (interfaces with the environment)
Let $\mathcal{I}_n : [n(n+1)] \to [n(n+1)]$, for every natural number $n \geq 1$, be the bijection:

$$\mathcal{I}_n(i) = \begin{cases} i(n+1) & \text{if } i \leq n \\ y_i & \text{otherwise} \end{cases}$$

where the values for $y_i$ are defined as follows: for every $n+1 \leq i \leq n(n+1)$, $y_i$ is the smallest number between 1 and $n(n+1)$ different from $y_j$, for all $j < i$.

We define the network $\mathtt{ITF}_n$ representing the bijection $\mathcal{I}_n$ as follows:

$$\mathtt{ITF}_n = \mathsf{I}_{n(n+1)} \circ (n(n+1), n(n+1), \mathtt{itf}_n) \circ \mathsf{I}_{n(n+1)}$$

where $\mathtt{itf}_n$ is defined by

$$\mathtt{itf}_n = ( \sum_{i \in [n(n+1)]} er_i(x) \; ; \underline{\underline{s}}_{\mathcal{I}(i)}(x)) * \underline{\underline{\delta}}$$

□

A network $\mathtt{ITF}_n^{-1}$ connecting the output port $n+1$ of the named component $N_{p(i)}$ to the output port $i$ of an SDL network containing $n$ named components, for each $i \in [n]$, can be defined analogously using the function $\mathcal{I}_n^{-1}$.

**Definition 7.5** (inter-connection operator)
For $n \geq 1$, we define the *inter-connection* operation $\amalg_n$, of arity

$$\underbrace{(((n+1) \to (n+1)) \times \ldots \times ((n+1) \to (n+1)))}_{n \text{ times}} \to (n \to n).$$

The operator $\amalg_n$ is defined by

$$\amalg_n(t_1, \ldots, t_n) = \mathsf{I}_n \circ ((\mathtt{ITF}_n \circ (t_1 \mathbin{+\!\!+} \ldots \mathbin{+\!\!+} t_n) \circ \mathtt{ITF}_n^{-1} \circ (\mathsf{I}_n \mathbin{+\!\!+} \mathtt{F}_n)) \uparrow^{n^2}) \circ \mathsf{I}_n$$

□

**Definition 7.6** (SDL network)
Let $n$ be the number of names in $\mathcal{C}$, and let $N_{p(1)}, \ldots, N_{p(n)}$ be the named components in $\mathcal{C}$. Then $\amalg_n(N_{p(1)}, \ldots, N_{p(n)})$ is an SDL network. □

In Figure 5, Def. 7.6 is illustrated by means of a graphical representation, for the case $n = 3$. We use the convention that the $i$-th entry and the $i$-th exit in the dotted feedback line correspond to the $i$-th feedback.

In the next subsection, we will introduce more abstract models derived from the process algebra model.

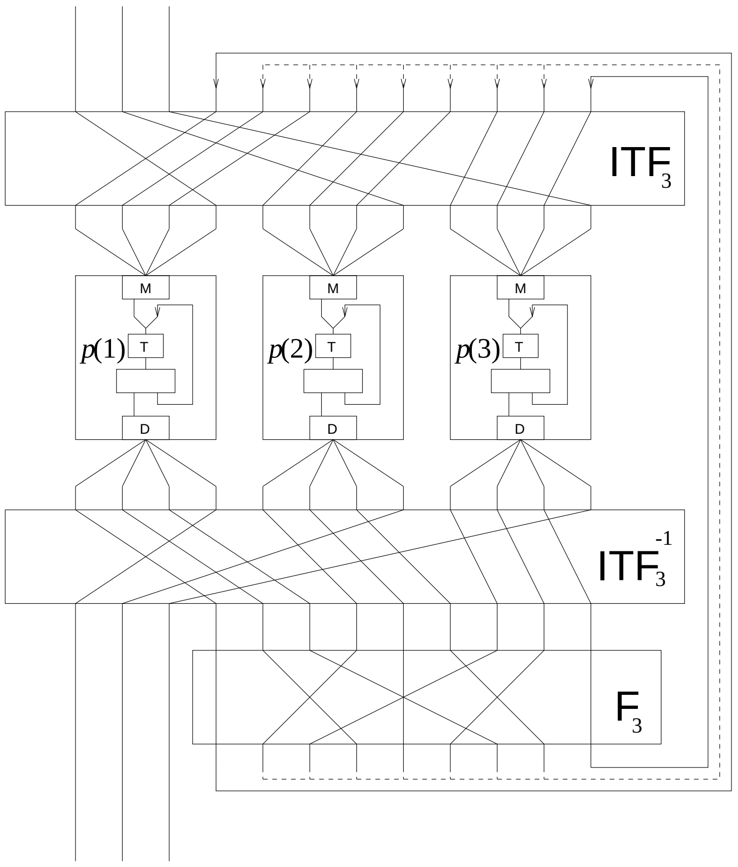

Figure 5: SDL network

### 7.3 Abstract semantics for SDL networks

In this section, more abstract models for SDL networks are derived from the process algebra model presented in Section 6.3 and their compositionality with respect to the inter-connection operation introduced in Section 7. The main result is that also in this case trace equivalence is fully-abstract with respect to history equivalence.

### Derivation of related models

In this subsection, the derivation of several models from the process algebra model $\mathsf{SDLProc}(\mathcal{C}, D)$ is described. We obtain these models by defining equivalences on SDL networks.

In order to be able to use models of process algebra in the derivation of the history model the input streams of a network have to be represented by networks. The resulting input networks are then composed with the original network. The input streams concerned contain data which are to be sent to the network, as well as $\sigma$s representing the time steps in between.

**Definition 7.7** (input network)
Let $\rho$ be a stream over $(\{\mathsf{nil}\} \times D) \cup \{\sigma\}$. The input network associated with $\rho$ is the network $\mathtt{SOURCE}_1(\rho) = (0, 1, \mathtt{source}_1(\rho))$ where

$$\begin{array}{l} \mathtt{source}_1(\rho) \;= \\ \qquad \delta \lhd |\rho| = 0 \rhd (\underline{\underline{s}}_1(hd(\rho)) \cdot \mathtt{source}_1(tl(\rho)) \lhd isd(hd(\rho)) \rhd \sigma_{\mathsf{rel}}(\mathtt{source}_1(tl(\rho)))) \end{array}$$

where $isd(d)$ yields true if $d$ is a datum in $(\{\mathsf{nil}\} \times D)$.

Let $f : m \to n$ be a network and $\rho_1, \ldots, \rho_m$ be streams. The network $f(\rho_1, \ldots, \rho_m)$ is defined by

$$f(\rho_1, \ldots, \rho_m) = (\mathtt{SOURCE}_1(\rho_1) + \!\!\!+ \ldots + \!\!\!+ \mathtt{SOURCE}_1(\rho_m)) \circ f$$

□

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 $P$. We consider as complete traces the union of the complete traces of $P$ as defined in [11], 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 $P$. We treat the time step $\sigma$ in these traces on the same footing as actions. Note however that the distinction between successful termination and deadlock/livelock made in such traces is irrelevant here because the processes modelling timed asynchronous dataflow networks do not include successfully terminating processes.

**Definition 7.8** (stream extraction)
Let $\beta$ be a trace over

$$\{s_i(d) \mid i \in [m], d \in (\{\mathsf{nil}\} \times D)\} \cup \{r_j(d) \mid j \in [n], d \in (\{\mathsf{nil}\} \times D)\} \cup \{\sigma\}.$$

We write $\mathsf{stream}_i^{in}(\beta)$ for the stream of data obtained by first removing all send actions 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 read actions and after that replacing each action of the form $s_i(d)$ by $d$. Often, we write only the relevant part from these pairs in $\{\mathsf{nil}\} \times D$. □

For a network $f : m \to n$ and an $m$-tuple of streams $(\rho_1, \ldots, \rho_m)$, the possible $n$-tuples of output streams can now be obtained from the traces of the process corresponding to the network $f(\rho_1, \ldots, \rho_m)$ using stream extraction.

**Definition 7.9** (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

$$[f](\rho_1, \ldots, \rho_m) \;=\; \{(\mathsf{stream}_1^{out}(\beta), \ldots, \mathsf{stream}_n^{out}(\beta)) \mid \beta \in \mathsf{trace}(f(\rho_1, \ldots, \rho_m))\}$$

□

**Definition 7.10** ($\equiv_{\text{history}}$)
The *history* equivalence $\equiv_{\text{history}}$ on timed asynchronous dataflow networks is defined by $f \equiv_{\text{history}} g$ iff $[f] = [g]$. □

Various interesting models for process algebra are obtained by defining equivalence relations on processes. We mention:

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

Branching bisimulation was introduced, in the setting of $\text{ACP}^{\tau}_{\text{drt}}$, in [5]. $P \equiv_{\text{ct}} Q$ iff $\mathsf{trace}(P) = \mathsf{trace}(Q)$. The above-mentioned equivalences on processes naturally induce corresponding equivalences on timed asynchronous dataflow networks, and consequently on SDL networks.

**Definition 7.11** ($\equiv_{\text{trace}}$)
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 $P \equiv_{\text{ct}} Q$. □

**Definition 7.12** ($\equiv_{\text{bisim}}$)
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 $P \leftrightarrows_{\text{b}} Q$. □

## 8 Conclusions

We conclude that an intuitively clear semantic model for SDL, including timer handling, can be based on a relative discrete time version of network algebra. The main ingredient of this model is a wire which, together with a priority mechanism in the sequential composition and feedback, allows immediate execution of timer resets.

We obtain networks which can be seen as SDL systems. The semantics of these networks can be expressed in terms of traces, and this trace semantics carries the minimal information in order to obtain compositionality for the constructor operator of our networks.

**Acknowledgements**

The third author acknowledges Bogdan Warinschi and Yaroslav Usenko for their helpful discussions and suggestions.

## 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 P9208b, 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 with abstraction. In H. Reichel, editor, *Fundamentals of Computation Theory*, pages 1–15. LNCS 965, Springer-Verlag, 1995.

[6] J.C.M. Baeten and J.A. Bergstra. Discrete time process algebra. *Formal Aspects of Computing*, 8:188–208, 1996.

[7] J.C.M. Baeten, J.A. Bergstra, and J.W. Klop. Syntax and defining equations for an interrupt mechanism in process algebra. *Fundamenta Informaticae*, 9:127–168, 1986.

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

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

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

[11] 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.

[12] J.A. Bergstra and C.A. Middelburg. Process algebra semantics of $\varphi$SDL. Research Report 68, United Nations University, International Institute for Software Technology, April 1996.

[13] J.A. Bergstra, C.A. Middelburg, and Gh. Ştefănescu. Network algebra for synchronous and asynchronous dataflow. Report P9508, University of Amsterdam, Programming Research Group, October 1995.

[14] J.A. Bergstra, C.A. Middelburg, and Gh. Ştefănescu. Network algebra for asynchronous dataflow. To appear in *International Journal of Computer Mathematics*, 1997.

[15] 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.

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

[17] 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.

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

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

[20] 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.

[21] S. Mauw. Example specifications in $\varphi$SDL. Computing Science Report 96-04, Eindhoven University of Technology, Department of Mathematics and Computing Science, 1996.

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

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

[24] Specification and description language (SDL). ITU-T Recommendation Z.100, Revision 1, 1994.

## A Proofs of main lemmas

In this appendix we prove Lemma 6.6 from Section 6.2 and Lemma 6.11 from Section 6.3.

**Lemma 6.6** *For the wire* $\mathsf{I}_1 = (1, 1, \mathtt{tsd}_1^1(\varepsilon))$, $\mathsf{I}_1 \circ \mathsf{I}_1 = \mathsf{I}_1$.

**Proof:** First we state some equalities which are useful for this proof.

If $y \neq \overset{\bullet}{\delta}$, we have

$$x \cdot \underline{\underline{\tau}} \cdot y = x \cdot y \qquad (*)$$

and

$$y + \underline{\underline{\delta}} = y$$

From axiom $DRTB2$ of $\mathrm{ACP}^{\tau}_{\mathrm{drt}}$, i.e.

$$x \cdot (\underline{\underline{\tau}} \cdot (y + \nu_{\mathsf{rel}}(z) + \underline{\underline{\delta}}) + y) = x \cdot (y + \nu_{\mathsf{rel}}(z) + \underline{\underline{\delta}}) \ ,$$

it follows that $x \cdot (\underline{\underline{\tau}} \cdot (y + z) + y) = x \cdot (y + z)$, if $z$ is of the form $z = \underline{\underline{a}} \cdot v$; and from this it follows that

$$\underline{\underline{\tau}} \cdot x = \underline{\underline{\tau}} \cdot (y + z) \Rightarrow \underline{\underline{\tau}} \cdot x = \underline{\underline{\tau}} \cdot (y + \underline{\underline{\tau}} \cdot x + z) \qquad (**)$$

if $z$ is of the form $z = \underline{\underline{a}} \cdot v$.

A wire $\mathsf{I}_1$ is a network $(1, 1, \mathtt{tsd}^1_1)$, where we have an input port denoted by 1 and an output port denoted by 1. For a calculation in $\mathrm{ACP}^{\tau}_{\mathrm{drt}}$, we need to distinguish the ports by their names, so for this proof we denote by $\mathtt{tsd}^i_j$ our process using the input port $i$ and the output port $j$. We define:

$$\begin{array}{rcl} P(\sigma_1, \sigma_2, \sigma_3) & = & \tau_{I(2,3)}(\partial_{H(2,3)}((\mathtt{tsd}^1_2(\sigma_1) \mathbin{|||} \mathtt{tsd}^3_4(\sigma_3)) \parallel \mathtt{tsd}^2_3(\sigma_2))) \\ Q(\sigma_1, \sigma_2, \sigma_3) & = & \mathtt{tsd}^1_4(\sigma_3 {}^\frown \sigma_2 {}^\frown \sigma_1) \end{array}$$

where $\sigma_i$ are sequences of data in $D$. The statement of the lemma becomes:

$$P(\varepsilon, \varepsilon, \varepsilon) = Q(\varepsilon, \varepsilon, \varepsilon)$$

It follows that

$P(\varepsilon, \varepsilon, \varepsilon) = er_1(x); P(x, \varepsilon, \varepsilon)$ and

$Q(\varepsilon, \varepsilon, \varepsilon) = er_1(x); Q(\varepsilon, \varepsilon, x) = er_1(x); Q(x, \varepsilon, \varepsilon)$ .

We apply RSP and finish the proof using $(*)$ and the fact:

$|\sigma_3 {}^\frown \sigma_2 {}^\frown \sigma_1| > 0 \ \Rightarrow \ \underline{\underline{\tau}} \cdot P(\sigma_1, \sigma_2, \sigma_3) = \underline{\underline{\tau}} \cdot Q(\sigma_1, \sigma_2, \sigma_3)$ .

To prove this, there are 7 cases to distinguish: for $\sigma_i$ empty or not, $i \in [3]$ and fulfilling the condition $|\sigma_3 {}^\frown \sigma_2 {}^\frown \sigma_1| > 0$. Each of these cases can be proved using RSP, but the guarded equation that both processes satisfy is different from case to case. We will discuss some relevant cases.

1. $|\sigma_3| > 0, |\sigma_1| = |\sigma_2| = 0$, only the third queue is non-empty. In this case we consider the guarded equation

   (1) $\quad X(\varepsilon, \varepsilon, \sigma_3) = \underline{\underline{\tau}} \cdot (\underline{\underline{er}}_1(x) \ ; X(x, \varepsilon, \varepsilon) + \underline{\underline{s}}_4 \cdot (hd(\sigma_3)) \cdot X(\varepsilon, \varepsilon, tl(\sigma_3)))$

   Both $\underline{\underline{\tau}} \cdot P(\varepsilon, \varepsilon, \sigma_3)$ and $\underline{\underline{\tau}} \cdot Q(\varepsilon, \varepsilon, \sigma_3)$ satisfy it.

2. $|\sigma_1| > 0, |\sigma_2| = |\sigma_3| = 0$, only the first queue is non-empty. In this case we consider the guarded equation

$$(2) \qquad X(\sigma_1, \varepsilon, \varepsilon) = \underline{\underline{\tau}} \cdot (\underline{\underline{er}}_1(x) \; ; X(\sigma_1 {}^\frown x, \varepsilon, \varepsilon) + \underline{\underline{\tau}} \cdot X(tl(\sigma_1), hd(\sigma_1), \varepsilon))$$

The equation 2 is guarded because a repeated replacing of the $\tau$-guarded variables by the right-hand side of their equations leads to the left-hand side variable from the equation 1, which is completely guarded. The process $\underline{\underline{\tau}} \cdot P(\sigma_1, \varepsilon, \varepsilon)$ satisfies the equation 2, which can easily be proved using $(*)$. The process $\underline{\underline{\tau}} \cdot Q(\sigma_1, \varepsilon, \varepsilon)$ satisfies the equation 2 as well:

$$\begin{array}{rcl}
\underline{\underline{\tau}} \cdot Q(\sigma_1, \varepsilon, \varepsilon) & = & \underline{\underline{\tau}} \cdot (\underline{\underline{er}}_1(x); Q(\sigma_1 {}^\frown x, \varepsilon, \varepsilon) + \underline{\underline{s}}_4(hd(\sigma_1)) \cdot Q(tl(\sigma_1), \varepsilon, \varepsilon)) \\
& \overset{DRTB2}{=} & \underline{\underline{\tau}} \cdot (\underline{\underline{er}}_1(x); Q(\sigma_1 {}^\frown x, \varepsilon, \varepsilon) + \\
& & \underline{\underline{\tau}} \cdot (\underline{\underline{er}}_1(x); Q(\sigma_1 {}^\frown x, \varepsilon, \varepsilon) + \underline{\underline{s}}_4(hd(\sigma_1)) \cdot Q(tl(\sigma_1), \varepsilon, \varepsilon))) \\
& = & \underline{\underline{\tau}} \cdot (\underline{\underline{er}}_1(x); Q(\sigma_1 {}^\frown x, \varepsilon, \varepsilon) + \underline{\underline{\tau}} \cdot Q(\sigma_1, \varepsilon, \varepsilon)) \\
& = & \underline{\underline{\tau}} \cdot (\underline{\underline{er}}_1(x); Q(\sigma_1 {}^\frown x, \varepsilon, \varepsilon) + \underline{\underline{\tau}} \cdot Q(tl(\sigma_1), hd(\sigma_1), \varepsilon))
\end{array}$$

3. $|\sigma_i| > 0, \; i \in [3]$, every sequence is non-empty. Then both $\underline{\underline{\tau}} \cdot P(\sigma_1, \sigma_2, \sigma_3)$ and $\underline{\underline{\tau}} \cdot Q(\sigma_1, \sigma_2, \sigma_3)$ are solutions for the guarded equation

$$(3) \qquad \begin{array}{rl}
X(\sigma_1, \sigma_2, \sigma_3) = & \underline{\underline{\tau}} \cdot (\underline{\underline{er}}_1(x); X(\sigma_1 {}^\frown x, \sigma_2, \sigma_3) + \\
& \underline{\underline{\tau}} \cdot X(tl(\sigma_1), \sigma_2 {}^\frown hd(\sigma_1), \sigma_3) + \\
& \underline{\underline{\tau}} \cdot X(\sigma_1, tl(\sigma_2), \sigma_3 {}^\frown hd(\sigma_2)) + \\
& \underline{\underline{s}}_4(hd(\sigma_3)) \cdot X(\sigma_1, \sigma_2, tl(\sigma_3)))
\end{array}$$

The system is guarded for similar reasons as in the previous case. The process $\underline{\underline{\tau}} \cdot Q(\sigma_1, \sigma_2, \sigma_3)$ satisfies it, which can be proved using $(**)$ twice. The process $\underline{\underline{\tau}} \cdot P(\sigma_1, \sigma_2, \sigma_3)$ satisfies it as well, which can be proved using $(*)$.

The other cases can be treated along the lines of the above ones. $\Box$

**Lemma 6.11** *For the wire* $\mathsf{I}_1 = (1, 1, \mathsf{ssd}_1^1(\varepsilon))$, $\mathsf{I}_1 \circ \mathsf{I}_1 = \mathsf{I}_1$.

**Proof:** A wire $\mathsf{I}_1$ is a network $(1, 1, \mathsf{ssd}_1^1)$, where we have an input port denoted by 1 and an output port denoted by 1. For a calculation in $\mathrm{ACP}^\tau_{\mathrm{drt},\theta}$, we need to distinguish the ports by their names, so for this proof we denote by $\mathsf{ssd}_j^i$ our process using the input port $i$ and the output port $j$. The following definitions will be useful in the proof:

$$\begin{array}{rcl}
P(\sigma_1, \sigma_2, \sigma_3) & = & \tau_{I(2,3)}(\theta(\partial_{H(2,3)}((\mathsf{ssd}_2^1(\sigma_1) \;|\!|\!|\; \mathsf{ssd}_4^3(\sigma_3)) \parallel \mathsf{ssd}_3^2(\sigma_2)))) \\
Q(\sigma_1, \sigma_2, \sigma_3) & = & \mathsf{ssd}_4^1(\sigma_3 {}^\frown \sigma_2 {}^\frown \sigma_1)
\end{array}$$

where $\sigma_i$ are sequences of data in $\mathcal{D}$. The statement of the lemma becomes

$$P(\varepsilon,\varepsilon,\varepsilon) = Q(\varepsilon,\varepsilon,\varepsilon)$$

We will use $\sigma'$ for the sequence $\sigma$ after the function *reset* is applied. For the legibility of the calculations we also introduce some auxiliary processes, namely

$$P'_y(\sigma_1,\sigma_2,\sigma_3) = \tau_{I(2,3)}(\theta(\partial_{H(2,3)}((\underline{\underline{s}}_2((\mathsf{reset},y))\cdot \mathtt{ssd}_2^1(\sigma'_1) \mathbin{|\!|\!|} \mathtt{ssd}_4^3(\sigma_3)) \parallel \mathtt{ssd}_3^2(\sigma_2))))$$

and with these notations we have

$$P(\varepsilon,\varepsilon,\varepsilon) \; = \; \textstyle\sum_{\substack{x\neq\mathsf{reset}\\ y\in D}} r_1((x,y))\cdot P((x,y),\varepsilon,\varepsilon)+$$

$$\textstyle\sum_{y\in D} r_1((\mathsf{reset},y))\cdot(\tau_{I(2,3)}(\theta(\partial_{H(2,3)}((\underline{\underline{s}}_2((\mathsf{reset},y))\cdot \mathtt{ssd}_2^1(\varepsilon) \mathbin{|\!|\!|} \mathtt{ssd}_4^3(\varepsilon)) \parallel \mathtt{ssd}_3^2(\varepsilon)))))$$

$$= \textstyle\sum_{\substack{x\neq\mathsf{reset}\\ y\in D}} r_1((x,y))\cdot P((x,y),\varepsilon,\varepsilon) + \sum_{y\in D} r_1((\mathsf{reset},y))\cdot P'_y(\varepsilon,\varepsilon,\varepsilon)$$

and for the right hand side we have

$$Q(\varepsilon,\varepsilon,\varepsilon) = \textstyle\sum_{y\in D} r_1(y)\cdot Q(y,\varepsilon,\varepsilon) + \sum_{\substack{x\neq\mathsf{reset}\\ y\in D}} r_1((x,y))\cdot Q((x,y),\varepsilon,\varepsilon) +$$

$$\textstyle\sum_{y\in D} r_1((\mathsf{reset},y))\cdot \underline{\underline{s}}_4((\mathsf{reset},y))\cdot Q(\varepsilon,\varepsilon,\varepsilon)$$

where $Q(\varepsilon,\varepsilon,(x,y)) = Q((x,y),\varepsilon,\varepsilon)$ was applied. Using Facts 1 and 2 below, we can apply RSP in order to finish the proof.

**Fact 1.** $\underline{\underline{\tau}}\cdot P(\sigma_1,\sigma_2,\sigma_3) = \underline{\underline{\tau}}\cdot Q(\sigma_1,\sigma_2,\sigma_3)$
for sequences $\sigma_1,\sigma_2,\sigma_3\in\mathcal{D}^\omega$ which do not contain data of the form $(\mathsf{reset},y)$.

**Fact 2.** $\underline{\underline{\tau}}\cdot P'_y(\varepsilon,\varepsilon,\varepsilon) = \underline{\underline{\tau}}\cdot \underline{\underline{s}}_4((\mathsf{reset},y))\cdot P(\varepsilon,\varepsilon,\varepsilon)$

**Proof:** We give the proof for the first fact. The second one can be proved as a particular case of the first.

The observation that none of the sequences can contain data of the form $(\mathsf{reset},y)$ is an important one. It expresses that when a datum of this form is received, it is delivered immediately.

The proof uses RSP showing that both processes $\underline{\underline{\tau}}\cdot P(\sigma_1,\sigma_2,\sigma_3)$ and $\underline{\underline{\tau}}\cdot Q(\sigma_1,\sigma_2,\sigma_3)$, for $|\sigma_3{}^\frown\sigma_2{}^\frown\sigma_1| > 0$, satisfy a guarded equation. Analogously with the proof of Lemma 6.6, there are several cases to distinguish, but we treat here only the most representative one. The other cases can be solved in a similar manner.

Now, for $|\sigma_i| > 0$, $i\in[3]$, both $\underline{\underline{\tau}}\cdot P(\sigma_1,\sigma_2,\sigma_3)$ and $\underline{\underline{\tau}}\cdot Q(\sigma_1,\sigma_2,\sigma_3)$ are solutions for the guarded equation

$$\begin{aligned}X(\sigma_1,\sigma_2,\sigma_3) &= \underline{\underline{\tau}}\cdot\big(\textstyle\sum_{y\in D}\underline{\underline{r}}_1((\mathsf{reset},y))\cdot\underline{\underline{s}}_4((\mathsf{reset},y))\cdot X(\sigma_1',\sigma_2',\sigma_3')+\\ &\textstyle\sum_{\substack{x\neq\mathsf{reset}\\ y\in D}}\underline{\underline{r}}_1((x,y))\cdot X(\sigma_1{}^\frown(x,y),\sigma_2,\sigma_3)+\\ &\underline{\underline{s}}_4(hd(\sigma_3))\cdot X(\sigma_1,\sigma_2,tl(\sigma_3))+\\ &\underline{\underline{\tau}}\cdot X(tl(\sigma_1),\sigma_2{}^\frown hd(\sigma_1),\sigma_3)+\underline{\underline{\tau}}\cdot X(\sigma_1,tl(\sigma_2),\sigma_3{}^\frown hd(\sigma_2))\big)\ .\end{aligned}\tag{4}$$

Equation 4 is guarded following the same arguments as in Lemma 6.6. The process $\underline{\underline{\tau}}\cdot Q(\sigma_1,\sigma_2,\sigma_3)$ satisfies this guarded equation because of (**) in the proof of Lemma 6.6.

for $y\neq\overset{\bullet}{\delta}$ and $z$ of the form $z=\underline{\underline{u}}\cdot v$, we have $\underline{\underline{\tau}}\cdot x=\underline{\underline{\tau}}\cdot(y+z)\Rightarrow\underline{\underline{\tau}}\cdot x=\underline{\underline{\tau}}\cdot(y+\underline{\underline{\tau}}\cdot x+\underline{\underline{\tau}}\cdot x+z)$.

We can write the process $P(\sigma_1,\sigma_2,\sigma_3)$ in the following form:

$$\begin{aligned}P(\sigma_1,\sigma_2,\sigma_3) &= \textstyle\sum_{y\in D}\underline{\underline{r}}_1((\mathsf{reset},y))\cdot P_y'(\sigma_1,\sigma_2,\sigma_3)+\\ &\textstyle\sum_{\substack{x\neq\mathsf{reset}\\ y\in D}}\underline{\underline{r}}_1((x,y))\cdot P(\sigma_1{}^\frown(x,y),\sigma_2,\sigma_3)+\\ &\underline{\underline{s}}_4(hd(\sigma_3))\cdot P(\sigma_1,\sigma_2,tl(\sigma_3))+\\ &\underline{\underline{\tau}}\cdot P(tl(\sigma_1),\sigma_2{}^\frown hd(\sigma_1),\sigma_3)+\underline{\underline{\tau}}\cdot P(\sigma_1,tl(\sigma_2),\sigma_3{}^\frown hd(\sigma_2))\end{aligned}$$

Then process $P(\sigma_1,\sigma_2,\sigma_3)$ satisfies the guarded equation 4 iff

$$\textstyle\sum_{y\in D}\underline{\underline{r}}_1((\mathsf{reset},y))\cdot P_y'(\sigma_1,\sigma_2,\sigma_3)=\sum_{y\in D}\underline{\underline{r}}_1((\mathsf{reset},y))\cdot\underline{\underline{s}}_4((\mathsf{reset},y))\cdot P(\sigma_1',\sigma_2',\sigma_3')$$

and this follow easily with RSP and $\underline{\underline{\tau}}\cdot P_y'(\sigma_1,\sigma_2,\sigma_3)=\underline{\underline{\tau}}\cdot\underline{\underline{s}}_4((\mathsf{reset},y))\cdot P(\sigma_1',\sigma_2',\sigma_3')$ .

We prove the last equality by expanding the left hand side term. Thus certain terms will be removed by $\partial_H$ or $\theta$; they will be denoted by (..). We further use the notation $P_y^i((x,y),\sigma)$ for $\underline{\underline{s}}_{i+1}((\mathsf{reset},y))\cdot\mathtt{ssd}_{i+1}^i(\sigma')\triangleleft x=\mathsf{reset}\triangleright\mathtt{ssd}_{i+1}^i(\sigma{}^\frown(x,y))$ .

$$\begin{aligned}&\underline{\underline{\tau}}\cdot P_y'(\sigma_1,\sigma_2,\sigma_3)\\ &=\underline{\underline{\tau}}\cdot\tau_{I(2,3)}(\theta(\partial_{H(2,3)}((\underline{\underline{s}}_2((\mathsf{reset},y))\cdot\mathtt{ssd}_2^1(\sigma_1')\mathbin{|||}\mathtt{ssd}_4^3(\sigma_3))\parallel\mathtt{ssd}_3^2(\sigma_2))))\\ &(\text{ we expand } \mathbin{|||}\ )\\ &=\underline{\underline{\tau}}\cdot\tau_{I(2,3)}(\theta(\partial_{H(2,3)}((\underline{\underline{s}}_2((\mathsf{reset},y))\cdot(\mathtt{ssd}_2^1(\sigma_1')\mathbin{|||}\mathtt{ssd}_4^3(\sigma_3))+\underline{\underline{er}}_3((x,y));(..)+\\ &\underline{\underline{s}}_4(hd(\sigma_3))(..))\parallel(\underline{\underline{er}}_2((x,y));P_y^2((x,y),\sigma_2)+\underline{\underline{s}}_3(hd(\sigma_2))(..)))))\\ &(\text{ expanding } \parallel \text{ and communicating })\\ &=\underline{\underline{\tau}}\cdot\tau_{I(2,3)}(\theta(\underline{\underline{c}}_2((\mathsf{reset},y))\cdot((\mathtt{ssd}_2^1(\sigma_1')\mathbin{|||}\mathtt{ssd}_4^3(\sigma_3))\parallel P_y^2((\mathsf{reset},y),\sigma_2))+\\ &\underline{\underline{s}}_4(hd(\sigma_3))(..)+\underline{\underline{c}}_3(hd(\sigma_2))(..)))\\ &(\ c_2((\mathsf{reset},y))\text{ has priority over the other two actions })\\ &=\underline{\underline{\tau}}\cdot\underline{\underline{\tau}}\cdot\tau_{I(2,3)}(\theta(\partial_{H(2,3)}((\mathtt{ssd}_2^1(\sigma_1')\mathbin{|||}\mathtt{ssd}_4^3(\sigma_3))\parallel\underline{\underline{s}}_3((\mathsf{reset},y))\cdot\mathtt{ssd}_3^2(\sigma_2'))))\end{aligned}$$

One can observe there is a symmetry between the formula from the first step and the last one; analogous calculations can go further in the same way obtaining

$$\begin{array}{l}
= \underline{\underline{\tau}} \cdot \underline{\underline{\tau}} \cdot \underline{\underline{\tau}} \cdot \tau_{I(2,3)}(\theta(\partial_{H(2,3)}((\mathtt{ssd}_2^1(\sigma_1') \mathbin{|||} \underline{\underline{s}}_4((\mathsf{reset}, y)) \cdot \mathtt{ssd}_4^3(\sigma_3')) \parallel \mathtt{ssd}_3^2(\sigma_2')))) \\
(\text{ now } s_4((\mathsf{reset}, y)) \text{ has priority over all the other actions }) \\
= \underline{\underline{\tau}} \cdot \underline{\underline{\tau}} \cdot \underline{\underline{\tau}} \cdot \underline{\underline{s}}_4((\mathsf{reset}, y)) \cdot P(\sigma_1', \sigma_2', \sigma_3') \\
= \underline{\underline{\tau}} \cdot \underline{\underline{s}}_4((\mathsf{reset}, y)) \cdot P(\sigma_1', \sigma_2', \sigma_3')
\end{array}$$

and we have finished the proof. $\square$