Grid Protocol Specifications *

Jan A. Bergstra[1,2] Alban Ponse[1]

[1] *University of Amsterdam, Programming Research Group*
Kruislaan 403, 1098 SJ Amsterdam, The Netherlands

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

E-mail: `janb@fwi.uva.nl - alban@fwi.uva.nl`

Abstract

A grid protocol models concurrent computation, and consists of one or more modules repeatedly performing parallel I/O and computation. We provide several concise specification formats and correctness results on (external) I/O behaviour, and illustrate our approach by examples.

Note: Some of the results described in this paper were published earlier in [BHP97]; other results were established by students in the '96/'97 course Process Algebra II delivered at the University of Amsterdam [BJM97], and in a master's thesis [Pou97].

1 Introduction

This paper surveys work done on specification and analysis of "grid protocols". It is based on the paper [BHP97], in which a simple class of these protocols is introduced. Furthermore, we survey two results on a larger class of these protocols. These results were obtained by various students [BJM97, Pou97]. For proofs we refer to the above mentioned references.

A *Grid protocol* models concurrent computation in a grid-like architecture. This type of protocols is based on Synchronous Concurrent Algorithms (SCAs) as developed by Tucker *et al.* [TT94]). Our motivation to follow this approach is illustrated by the following citation [TT94]:

> "many specialised models of computation possess the essential features of SCAs, including systolic arrays, neural networks, cellular automata and coupled map lattices. The parallel algorithms, architectures and dynamical systems that comprise the class SCAs have many applications, ranging from their use in special purpose devices [...] to computational models of biological and physical phenomena."

A grid protocol consists of *modules*, i.e., data processing units that can cooperate with each other or the environment by passing values (terminology is taken from [TT94].) This cooperation can be modeled in various ways. In this paper we consider value-passing by synchronization (communication actions). A module has a fixed functional device, and fixed incoming and outgoing channels or ports. Furthermore, it always has a current value, which is either initial or computed from the values received via its input ports. A channel models the connection with either one of the network's modules, or

*partially sponsored by Esprit Working Group 8533 NADA — New Hardware Design Methods.

with some external device. In terms of behaviour, a module repeatedly performs *parallel* execution of input and output actions, each of which operates on a distinct channel. Having executed all its I/O actions (Input/Output), the module updates its current value by application of its function to the newly received value(s). As an example, consider the module M in Fig. 1.

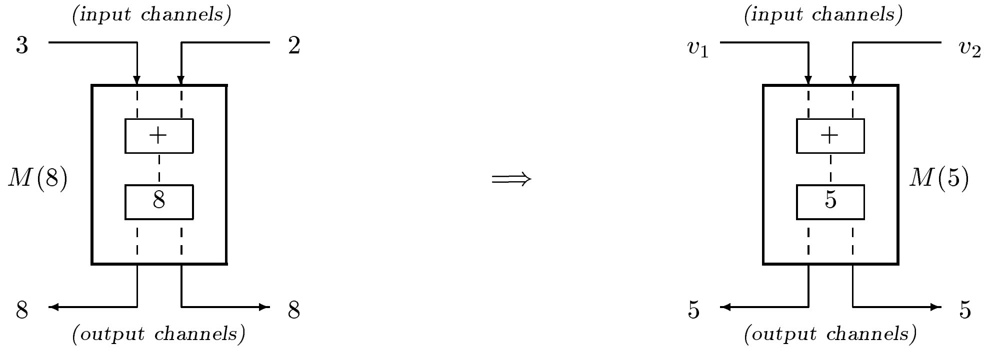


Figure 1: Example of the operation of a single module.

In this figure, the idea is that module M is regarded as having current value 8, notation $M(8)$, which can be sent along the two output channels, while values 3 and 2 can be received along the two input channels. Throughout the paper, we stick to the convention that modules are depicted by rectangular blocks, with input channels coming in at the top and output channels leaving from the bottom. The functionality of the example module is to add the values received, thus the next current value will be 5 (available at the output channels), and new values v_1 and v_2 can be received. After this parallel I/O behaviour, displayed at the left-hand side, the module has evolved into $M(5)$. A straightforward, recursive specification of this behaviour is

$$M(d) = (\| \text{ I/O value-passing actions}) \cdot M(e+f)$$

in case values e and f are received as new input values.

A grid protocol can be a network of processors or (groups of) points of measure in some physical phenomenon, for example a hardware device or a vibrating string. We come up with two characterization results on the external behaviour of grid protocols:

1. In the case that all modules and internal channels of the network form a connected graph and the external behaviour is located at *one* single module, we obtain a simple characterization result: the order of the (internal) synchronizations is not relevant and the network's external behaviour — *stream* transformation or generation — is determined by a prefix of I/O actions, followed by a simultaneous value update. This result is established in [BHP97].

2. In the case that all modules are synchronized by a device that keeps the modules in pace, the operation of the network is characterized by a prefix of (external) I/O actions, followed by a simultaneous value update. In this case, also the external activity of the network is transformation (or generation) of parallel streams, irrespective of location of I/O and connectedness. This result is established in [BJM97, Pou97].

We further motivate our approach as one that yields an operational perspective on the *module* level, i.e., value-passing by arbitrary interleaved synchronizations, and that relates this perspective with a correctness characterization of a network's external I/O behaviour. Our approach is based on a combination of value-passing calculus CCS (Calculus of Communicating Systems, Milner [Mil89]), and process algebra ACP (Algebra of Communicating Processes, Bergstra and Klop [BK84, BK85, BW90]). Main ingredients are *early read* actions, in which a variable can get instantiated via communication (value-passing), and the *process prefix*, a generalization of Milner's action prefix which provides a means for binding variables in CCS. With these ingredients, a concise notation of parallel input is possible. In [BB94], Baeten and Bergstra proposed axiom systems for action prefix, process prefixes, and early read actions in the setting of ACP. These axiom systems are based on a *finite* data type. In order to specify and analyze grid protocols processing on infinite data types, we extend the process prefixing mechanism of [BB94] to an infinitary setting. For specification of data and value-passing we refer to μCRL [GP95], an ACP-based approach in which both data, processes and their combined use can be formally specified and analyzed.

After a brief introduction to the axiom system $\mathrm{ACP}^{\tau}(A,\gamma)$, no-exit iteration and alphabet axioms (Section 2), the system $\mathrm{ACP}^{\tau}_{er}(A,\gamma)$ is defined. In this system, $\mathrm{ACP}^{\tau}(A,\gamma)$ is combined with process prefixes and early read actions (Section 3). In Section 4, we give a specification format for *modules* in finite, connected networks with I/O located at one port. We recall a simple equation that characterizes external behaviour, and hence two *correctness*-properties of the specification. In Section 5 we introduce a second class of grid protocols: *Beating Grid Protocols*. These networks are controlled by a *Beat*-process, i.e., a synchronization device that keeps parallel I/O of the whole network in phase. For this class we consider two types of *Beat*-processes, and state a general characterization result. Finally, in Section 6 we consider as an example an approximation of solutions of the one-dimensional *wave equation*, which can be modeled either in a connected grid protocol, or a beating grid protocol.

Acknowledgements. Jos van Wamel and Gerard Kok are thanked for many contributions and discussions, especially concerning Section 6.1. Furthermore, Egbert-Jan van Buiten, Merijn de Jonge, Ramin Monajemi, and Marco Pouw are thanked for their contributions in establishing specification and verification of beating grid protocols.

2 Process Algebra: Operations, Axioms, and Rules

In this section we recall some basic process algebra (without explicit use of data): the system $\mathrm{ACP}^{\tau}(A,\gamma)$, standard concurrency, and no-exit iteration. Furthermore, we introduce generalized merge, expansion and alphabet axioms, all of which are essential for specification (and verification) of grid protocols.

2.1 $\mathrm{ACP}^{\tau}(A,\gamma)$, Standard Concurrency, and No-Exit Iteration

The process algebraic framework $\mathrm{ACP}^{\tau}(A,\gamma)$ (ACP with branching bisimulation) has two parameters: a set A of constants modeling atomic actions, and a (partial) binary, commutative and associative *communication function* γ on A, defining which actions communicate. Furthermore there are constants δ (deadlock or inaction) and τ (silent step). Process operations in $\mathrm{ACP}^{\tau}(A,\gamma)$ are alternative composition or choice ($+$), sequential composition ($\cdot$), parallel composition or merge ($\|$), left and communication merge ($\lfloor\!\lfloor$ and $|$, used for the axiomatization of $\|$), encapsulation (∂_H), and hiding (τ_I). We mostly suppress the $\cdot$ in process expressions, and brackets according to the following precedences: $\cdot > \{\|, \lfloor\!\lfloor, |\} > +$. Process expressions are subject to the axioms of $\mathrm{ACP}^{\tau}(A,\gamma)$, displayed in Table 1 ($x, y, z, \ldots$ ranging over processes). Note that $+$ and $\cdot$ are associative.

We provide a slightly modified version of ACP (i.e., the axioms A1–7, CF1,2 and CM1–8), comprising commutativity of $\|$ and $|$, defined by the (new) axiom CMC (Communication Merge is Commutative).

Table 1: The axioms of $\mathrm{ACP}^\tau(A,\gamma)$ and for no-exit iteration, where $a,b \in A_{\delta,\tau}$, $H, I \subseteq A$.

(A1)	$x + y = y + x$	(B1)	$x\tau = x$
(A2)	$x + (y + z) = (x + y) + z$	(B2)	$x(\tau(y + z) + y) = x(y + z)$
(A3)	$x + x = x$		
(A4)	$(x + y)z = xz + yz$		
(A5)	$(xy)z = x(yz)$		
(A6)	$x + \delta = x$		
(A7)	$\delta x = \delta$		
(CF1)	$a \mid b = \gamma(a,b)$ if $\gamma(a,b)\downarrow$		
(CF2)	$a \mid b = \delta$ otherwise		
(CM1)	$x \parallel y = (x \mathbin{⌊\!⌊} y + y \mathbin{⌊\!⌊} x) + x \mid y$	(D1)	$\partial_H(a) = a$ if $a \notin H$
(CM2)	$a \mathbin{⌊\!⌊} x = ax$	(D2)	$\partial_H(a) = \delta$ if $a \in H$
(CM3)	$ax \mathbin{⌊\!⌊} y = a(x \parallel y)$	(D3)	$\partial_H(x + y) = \partial_H(x) + \partial_H(y)$
(CM4)	$(x + y) \mathbin{⌊\!⌊} z = x \mathbin{⌊\!⌊} z + y \mathbin{⌊\!⌊} z$	(D4)	$\partial_H(xy) = \partial_H(x) \cdot \partial_H(y)$
(CMC)	$x \mid y = y \mid x$	(TI1)	$\tau_I(a) = a$ if $a \notin I$
(CM5)	$ax \mid b = (a \mid b)x$	(TI2)	$\tau_I(a) = \tau$ if $a \in I$
(CM7)	$ax \mid by = (a \mid b)(x \parallel y)$	(TI3)	$\tau_I(x + y) = \tau_I(x) + \tau_I(y)$
(CM8)	$(x + y) \mid z = x \mid z + y \mid z$	(TI4)	$\tau_I(xy) = \tau_I(x) \cdot \tau_I(y)$
(NEI1)	$x^\omega = x \cdot (x)^\omega$	(NEI3)	$\partial_H(x^\omega) = (\partial_H(x))^\omega$
(RSP^ω)	$x = y \cdot x \implies x = y^\omega$	(NEI4)	$\tau_I(x^\omega) = (\tau_I(x))^\omega$

As a consequence, the symmetric version of CM5 and CM8, i.e. CM6 and CM9 respectively, are left out (cf. [BW90]). Furthermore, we adopt associativity of $\parallel$ and $\mid$. Commutativity and associativity of these operations is known as Standard Concurrency [BT84], and is referred to as SC.

In this paper we only consider two-party communication or *handshaking*, axiomatized by

$$x \mid y \mid z = \delta$$

(see [BT84]). Some informal explanation. Often, $+$ is used as an operation facilitating analysis rather than as a specification primitive: concurrency is analyzed in terms of sequential composition, choice and communication. Verification of a concurrent system $\partial_H(C_1 \parallel ... \parallel C_n)$ generally boils down to representing the possible executions with $+$ and $\cdot$, having applied left-merge ($⌊\!⌊$), communication merge, and encapsulation (∂_H, by which communication between components C_i can be enforced). After renaming internal activity to the silent, unobservable action τ with help of the hiding operator τ_I (also called 'abstraction'), this may yield a simple and informative specification of external behaviour. For a detailed introduction to $\mathrm{ACP}^\tau(A,\gamma)$ and SC we refer to [BW90].

In order to describe iterative, non-terminating processes we shall use the unary operation $\_^\omega$, *perpetual loop* or *no-exit iteration*, for which NEI1 in Table 1 is the defining axiom. This operation is introduced by Fokkink in [Fok97]. In that paper, several completeness results are established, among which the fact that BPA (axioms A1–A5) with NEI1 and RSP$^\omega$ characterizes strong bisimilarity. (The 'missing' axiom NEI2 concerns the empty process ϵ, and reads $(x+\epsilon)^\omega = x^\omega$.) It should be remarked that RSP$^\omega$ is not sound in the setting with the silent step τ. For example, each process $\tau \cdot P$ satisfies

$$\tau \cdot P = \tau \cdot \tau \cdot P,$$

though $\tau \cdot P = \tau^\omega$ is of course a very undesirable identity.

2.2 Generalized Merge, Expansion, and Alphabet Axioms

The *generalized merge*

$$\left[\mathop{\|}_{i\in I} P_i\right]$$

abbreviates the expression $(P_{i_1} \parallel P_{i_2} \parallel \ldots \parallel P_{i_n})$ for $I = \{i_1, i_2, \ldots, i_n\}$ a non-empty, finite set of indices. This notation can be justified by commutativity and associativity of $\parallel$ (SC). It is used to describe grid protocols as the concurrent composition of a finite number of modules. If I is a singleton, say $I = \{i_1\}$,

$$\left[\mathop{\|}_{i\in\{i_1\}} P_i\right] = P_{i_1}.$$

We often write

$$\left[\mathop{\|}_{i=1}^{n} P_i\right] \text{ rather than } \left[\mathop{\|}_{i\in\{1,\ldots,n\}} P_i\right].$$

The following result follows easily by induction on n, exploiting $\tau x \mathbin{\|\!\_} y = \tau(x \parallel y)$ and $x\tau = x$. Though exposition of proofs and supporting lemmas is not in the scope of this paper, it makes sense to recall a number of additional primitives and axioms as this illustrates the flavour of process algebraic reasoning.

Lemma 2.2.1 (*Merge Lemma*). $\mathrm{ACP}^\tau(A,\gamma) + \mathrm{SC} \vdash x\left(\left[\mathop{\|}_{i=1}^{n} \tau y_i\right] \parallel z\right) = x\left(\left[\mathop{\|}_{i=1}^{n} y_i\right] \parallel z\right).$

In the setting of handshaking we can use the *Expansion Theorem* (cf. [BW90]):

$$\textit{Expansion}\text{: for } n \geq 3: \quad \left[\mathop{\|}_{i=1}^{n} P_i\right] = \sum_{j=1}^{n} P_j \mathbin{\|\!\_} \left[\mathop{\|}_{i\in\{1,\ldots,n\}\setminus\{j\}} P_i\right] + \sum_{j=2}^{n}\sum_{k=1}^{j-1} (P_j \mid P_k) \mathbin{\|\!\_} \left[\mathop{\|}_{i\in\{1,\ldots,n\}\setminus\{j,k\}} P_i\right].$$

Many proofs depend on application of *alphabet axioms*. Under certain conditions the scope, or the action set of τ_I or ∂_H applications can be changed, depending on the alphabet of a process. In Table 2 we give some axioms to determine the alphabet of a process P, notation $\alpha(P)$. Except for AB6, these axioms stem from [BBK87].

Starting from the alphabet of a process, the *conditional alphabet axioms* in Table 3 (also taken from [BBK87]) give conditions for changing either scope or action sets I, H of τ_I and ∂_H applications. Here $B \mid C$ for $B, C \subseteq A$ denotes the set $\{a \in A \mid a = \gamma(b,c) \text{ for some } b \in B, c \in C\}$.

Table 2: Alphabet axioms, $a \in A$.

(AB1)	$\alpha(\delta)$	$=$	$\emptyset = \alpha(\tau)$	|	(AB4)	$\alpha(ax)$	$=$	$\{a\} \cup \alpha(x)$
(AB2)	$\alpha(a)$	$=$	$\{a\}$	|	(AB5)	$\alpha(x+y)$	$=$	$\alpha(x) \cup \alpha(y)$
(AB3)	$\alpha(\tau x)$	$=$	$\alpha(x)$	|	(AB6)	$\alpha(x^\omega)$	$=$	$\alpha(x)$

Table 3: Conditional alphabet axioms, $H, I \subseteq A$.

(CA1)	$\alpha(x) \mid (\alpha(y) \cap H) \subseteq H$	$\Longrightarrow$	$\partial_H(x \parallel y)$	$=$	$\partial_H(x \parallel \partial_H(y))$
(CA2)	$\alpha(x) \mid (\alpha(y) \cap I) = \emptyset$	$\Longrightarrow$	$\tau_I(x \parallel y)$	$=$	$\tau_I(x \parallel \tau_I(y))$
(CA3)	$\alpha(x) \cap H = \emptyset$	$\Longrightarrow$	$\partial_H(x)$	$=$	x
(CA4)	$\alpha(x) \cap I = \emptyset$	$\Longrightarrow$	$\tau_I(x)$	$=$	x

3 Data and Process Prefixing

In this section we explain value-passing, which is the basic communication mechanism in grid protocols. Furthermore, we introduce early reads and process prefixing, and show that these form a concise notation for the mechanics of parallel value-passing.

3.1 Value-Passing

As grid protocols process data, we demand computability and decidability of all data involved (in the sense of [BT95]). We adopt a simple specification paradigm for data and actions parameterized with data, which originates from μCRL [GP95]. Data are used in *data-parametric sums*, and in *communications*.

In order to specify value-passing, let i, j be channel identifiers. Action $r_i(t)$ models *receiving* the particular value t along channel i. Action $s_j(t)$ models the *sending* of a data value t along port j. We assume that the communications defined in Table 4 are the only ones defined, so the handshaking paradigm is satisfied. We thus have a framework with data-parametric actions.

Table 4: Send-read communication for value-passing, $a, b \in A$.

$$a \mid b = \begin{cases} c_i(t) & \text{if } \{a,b\} = \{r_i(t), s_i(t)\} \\ \delta & \text{otherwise.} \end{cases}$$

Let for instance r_i be typed as an action that can carry values of type $\mathbb{N}$ (the natural numbers). So $r_i(0), r_i(1), \ldots$ are considered actions. An example of a data-parametric sum is the expression

$$\sum(v : \mathbb{N}, r_i(v)),$$

denoting a process that for an arbitrary value n of $\mathbb{N}$ can once perform $r_i(n)$, after which it terminates. A typical use of this construct is

$$\textstyle\sum(v : \mathbb{N}, r_i(v) \cdot s_j(v + v))$$

which represents the infinite summation

$$r_i(0) \cdot s_j(0) + r_i(1) \cdot s_j(2) + r_i(2) \cdot s_j(4) + \cdots .$$

Note that the type of the variable v is declared *in* the scope of the $\sum$-operation. For the $\sum$-operation, axioms and a proof rule are defined in [GP91c, GP94b]. In particular, these comprise α-conversion and axioms to change its scope. We adopt the convention that $\sum(\_, \_)$ binds strongest of all operations, for example

$$\textstyle\sum(v : \mathbb{N}, a(v))^\omega = (\sum(v : \mathbb{N}, a(v)))^\omega .$$

With help of send-read communication and the encapsulation operations ∂_H, one can easily model value-passing (cf. [Mil89]). For a small, typical example consider

$$R \;=\; \textstyle\sum(v : \mathbb{N},\ r_1(v) \cdot s_2(v + 1))^\omega,$$

a process that is willing to receive any natural along channel 1, and send its successor along channel 2. Assume that R is executed in parallel with

$$s_1(5) \cdot S,$$

a process that initially sends the value 5 along channel 1. The value-passing of 5 between these two can be represented by

$$\partial_{\{r_1,s_1\}}(R \parallel s_1(5) \cdot S)$$

where we adopt the notation $\partial_{\{r_1,s_1\}}$, only mentioning the identifiers r_1, s_1, from μCRL. Hence, single $r_1(n)$ and $s_1(n)$ actions cannot occur and are thus enforced to communicate. We derive

$$\begin{array}{rcl} \partial_{\{r_1,s_1\}}(R \parallel s_1(5) \cdot S) & \overset{\text{BKS1}}{=} & \partial_{\{r_1,s_1\}}(\sum(v : \mathbb{N},\ r_1(v) \cdot s_2(v + 1)) \cdot R \parallel s_1(5) \cdot S) \\ & = & c_1(5) \cdot \partial_{\{r_1,s_1\}}(s_2(6) \cdot R \parallel S) \end{array}$$

where the second identity follows from the axioms of $\mathrm{ACP}^\tau(A, \gamma)$ and those for the $\sum$-operation. By encapsulation, the action $s_1(5)$ enforces the communication action $c_1(5)$. So, the *value-passing* of 5 along channel 1 between the two parallel components of $\partial_{\{r_1,s_1\}}(R \parallel s_1(5) \cdot S)$ is modeled by the execution of the communication action $c_1(5)$. The resulting process is $\partial_{\{r_1,s_1\}}(s_2(6) \cdot R \parallel S)$. In the setting of μCRL, a detailed treatment of this value-passing format can be found in [GK95].

3.2 Parallel Input: Early Reads and Process Prefixing

Let D be some data type. We introduce the binary operation *process prefix* and *early read actions* as a means to provide concise notation for parallel input. Let i be a channel or port identifier, and v a variable of type D. Then

$$er_i(v); x = \sum(v : D, r_i(v) \cdot x)$$

is the axiom scheme that introduces the early read action $er_i(v)$ and the process prefix ;. This axiom reflects Milner's translation of the basic CCS term $a(x).E$ into the value-passing CCS term $\sum_{v \in V} a_v.\widehat{E\{v/x\}}$ where V is the value set and $\widehat{\ }$ the translation function [Mil89]. It is a μCRL-like interpretation of the early read axiom in [BB94]. It is meant that v may occur in process x. So

$$er_i(v); s_j(v) = \sum(v : D, r_i(v) \cdot s_j(v))$$

is an expression without free data-variables, and so is

$$er_i(v); s_j(t) = \sum(v : D, r_i(v) \cdot s_j(t))$$

for t a closed term of type D.

Let A_{er} be the extension of A (the set of atomic actions) with early read actions for any action $r_i : D_1 \times ... \times D_n$ declared over A. Axioms for process prefixing are given in Table 5. The axiom PP4 is considered to be parameterized with the type of the r_i action. Note that for the er actions we use *globally* typed variables.

Table 5: Early input and process prefixing, $a \in A$.

(PP1)	$\delta; x$	$= \delta$	|	(PP4)	$er_k(v); x$	$= \sum(v : D,\ r_k(v) \cdot x)$
(PP2)	$\tau; x$	$= \tau \cdot x$	|	(PP5)	$(x + y); z$	$= x; z + y; z$
(PP3)	$a; x$	$= a \cdot x$	|	(PP6)	$(x \cdot y); z$	$= x; (y; z)$

Let $\text{ACP}^{\tau}_{er}(A, \gamma)$ be the extension of $\text{ACP}^{\tau}(A, \gamma)$ with early read actions and process prefixes as introduced above. By the send-read communication paradigm (see Table 4) is that $er_i(v) \mid a = \delta$ for all $a \in A_{er}$. This is used in the following example, in which parallel input is unraveled (v, w, F typed as above):

$$\begin{aligned}
&(er_1(v) \parallel er_2(w)); s_l(F(v, w)) \\
&= (er_1(v) \mathbin{\lfloor\!\lfloor} er_2(w) + er_2(w) \mathbin{\lfloor\!\lfloor} er_1(v) + er_1(v) \mid er_2(w)); s_l(F(v, w)) \\
&= (er_1(v) \cdot er_2(w) + er_2(w) \cdot er_1(v) + \delta); s_l(F(v, w)) \\
&= (er_1(v) \cdot er_2(w)); s_l(F(v, w)) + (er_2(w) \cdot er_1(v)); s_l(F(v, w)) \\
&= er_1(v); (er_2(w); s_l(F(v, w))) + er_2(w); (er_1(v); s_l(F(v, w))).
\end{aligned}$$

Applying axiom PP4 to the last expression yields that

$$(er_1(v) \parallel er_2(w)); s_l(F(v, w)) = \left\{ \begin{array}{l} \sum(v : D,\ r_1(v) \cdot \sum(w : D,\ r_2(w) \cdot s_l(F(v, w)))) \\ + \\ \sum(w : D,\ r_2(w) \cdot \sum(v : D,\ r_1(v) \cdot s_l(F(v, w)))), \end{array} \right.$$

showing conciseness of notation with early read actions and process prefix. Observe that in case we replace w by v,

$$(er_1(v) \parallel er_2(v)); s_l(F(v,v)) = \begin{cases} \sum(v:D,\ r_1(v) \cdot \sum(v:D,\ r_2(v) \cdot s_l(F(v,v)))) \\ + \\ \sum(v:D,\ r_2(v) \cdot \sum(v:D,\ r_1(v) \cdot s_l(F(v,v)))). \end{cases}$$

This example models *non-deterministic* input along one of the channels $1, 2$, yielding send-action $s_l(F(k,k))$ with k being the value received. In the peculiar case that the channel identifier 2 is replaced by 1, we find that

$$(er_1(v) \parallel er_1(w)); s_l(F(v,w)) = \begin{cases} \sum(v:D,\ r_1(v) \cdot \sum(w:D,\ r_1(w) \cdot s_l(F(v,w)))) \\ + \\ \sum(w:D,\ r_1(w) \cdot \sum(v:D,\ r_1(v) \cdot s_l(F(v,w)))). \end{cases}$$

models the sending of either $s_l(F(j,k))$ or $s_l(F(k,j))$ if values j and k are sent along 1. Finally,

$$(er_1(v) \parallel er_1(v)); s_l(F(v,v)) = (er_1(v) \cdot er_1(v)); s_l(F(v,v))$$

describes the situation in which the first inputed value along channel 1 is neglected, and the second one instantiates $F(v,v)$.

Remark 3.2.1. Notice that the process $\sum(v:D, \sum(w:D,\ (r_1(v) \parallel r_2(w))s_l(F(v,w))))$ does *not* model parallel input: upon reading the first value along channel 1 or 2, the choice for the second read action is already fixed.
End remark.

Furthermore, τ_I and ∂_H-applications also apply to er-actions via axiom PP4, using the μCRL axioms that state that these applications commute with the $\sum$-operation. For instance,

$$\begin{aligned} \partial_{\{r_1\}}(er_1(v); P_1 + er_2(w); P_2) &= \partial_{\{r_1\}}(\textstyle\sum(v:D,\ r_1(v)P_1)) + \partial_{\{r_1\}}(\textstyle\sum(w:\mathbb{N},\ r_2(w)P_2)) \\ &= \textstyle\sum(v:D,\ \partial_{\{r_1\}}(r_1(v)P_1)) + \textstyle\sum(w:\mathbb{N},\ \partial_{\{r_1\}}(r_2(w)P_2)) \\ &= \textstyle\sum(v:D,\ \delta) + \textstyle\sum(w:\mathbb{N},\ r_2(w)\partial_{\{r_1\}}P_2)) \\ &= \delta + er_2(w); \partial_{\{r_1\}}(P_2) \\ &= er_2(w); \partial_{\{r_1\}}(P_2). \end{aligned}$$

4 Modules and Connected, single-I/O Networks

In this section we propose a specification format for *connected networks*. Such a network consists of *modules*, elementary data processing units which may have feedback. Next we introduce *connected networks* as a format for the parallel execution of such modules. Our modeling is based on [TT94], in which SCAs (Synchronous Concurrent Algorithms) are analyzed.

4.1 Modules

A module M_i has a (current) value d, a fixed (positive) number n of input channels $i_1, ..., i_n$, and a fixed (positive) number m of output channels $o_1, ..., o_m$. Channels have unit bandwidth and are unidirectional; this corresponds with our format for value-passing as discussed in the previous section.

We first a setting with only one data type D. Computation in M_i is modeled by a (total) value function $F_i : D^n \to D$. The complete operation of module $M_i(d)$ can be described as follows:

$$M_i(d) = \left(\left[\mathop{\|}_{j=1}^{n} er_{i_j}(v_j) \right] \parallel \left[\mathop{\|}_{j=1}^{m} s_{o_j}(d) \right] \right) ; M(F_i(v_1, ..., v_n)). \tag{1}$$

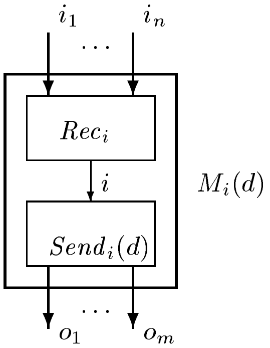
Unfortunately, this definition presupposes that M_i has no *feedback*, i.e. that $\{i_1, ..., i_n\} \cap \{o_1, ..., o_m\} = \emptyset$ because early read actions do not communicate (otherwise, the value in question would get lost). So for the particular case that M_i also has a feedback channel f, its definition should be something like

$$M_i(d) = \left(\left[\mathop{\|}_{j=1}^{n} er_{i_j}(v_j) \right] \parallel \left[\mathop{\|}_{j=1}^{m} s_{o_j}(d) \right] \right) ; M(F_i(d, v_1, ..., v_n)), \tag{2}$$

where the first argument of F_i models the feedback. This has two disadvantages: we lose uniformity of specification, and the feedback action is not explicit, only its effect is. As can be expected, we allow at most one feedback channel per module.

Because uniformity of specification is particularly useful in proving correctness, we adopt a "low level specification format" of modules, and obtain (1) as a derivable identity. Because feedback is an explicit, but hidden action in our specification format, we postpone the derivable variant of (2) till after the introduction of the operation of *networks*. Module $M_i(d)$ as described above is defined by means of two iterative processes. The first one of these defines the *receive*-part Rec_i of the module (modeling its read actions), the second its *send*-part $Send_i(d)$ (ready to send the value d along the ports $o_1, ..., o_m$). These two parts communicate along some channel i, internal to module M_i, and are also used to model computation of F_i. More precisely, communication of a value $F_i(\vec{d})$ by (internal) action $c(F_i(\vec{d}))$ can only take place if all parallel read actions of Rec_i have been executed, and if also $Send_i(d)$ has performed all its (parallel) send actions $s_{o_j}(d)$. This yields the following specification and picture of $M_i(d)$:

$$\begin{aligned}
M_i(d) &= \tau_{\{c_i\}} \circ \partial_{\{r_i, s_i\}}(Rec_i \parallel Send_i(d)), \\
Rec_i &= \left(\left[\mathop{\|}_{j=1}^{n} er_{i_j}(v_j) \right] ; s_i(F_i(v_1, ..., v_n)) \right)^{\omega}, \\
Send_i(d) &= \left[\mathop{\|}_{j=1}^{m} s_{o_j}(d) \right] \cdot Send_i, \\
Send_i &= \left(er_i(v); \left[\mathop{\|}_{j=1}^{m} s_{o_j}(v) \right] \right)^{\omega}.
\end{aligned}$$



In case M_i has no feedback, i.e., $\{i_1, ..., i_n\} \cap \{o_1, ..., o_m\} = \emptyset$, it follows that $M_i(d)$ has a process prefix

$$(er_{i_1}(v_1) \parallel ... \parallel er_{i_n}(v_n) \parallel s_{o_1}(d) \parallel ... \parallel s_{o_m}(d))$$

(this is a consequence of Theorem 4.3.3). After having read certain values $d_1, d_2, ..., d_n$ along channels $i_1, ..., i_n$, and having sent d along ports $o_1, ..., o_m$, the module's current value is updated to $F_i(d_1, ..., d_n)$ by a value-passing communication along channel i, renamed into the silent action τ. After this, the next process prefix is ready to be performed:

$$(er_{i_1}(v_1) \parallel ... \parallel er_{i_n}(v_n) \parallel s_{o_1}(F_i(d_1, ..., d_n)) \parallel ... \parallel s_{o_m}(F_i(d_1, ..., d_n)))\,.$$

For readability, we introduce the following abbreviation for synchronization and abstraction over some port i: we further write

$$P \parallel_i Q \quad \text{instead of} \quad \tau_{\{c_i\}} \circ \partial_{\{r_i, s_i\}}(P \parallel Q).$$

Henceforth, $M_i(d) = \mathit{Rec}_i \parallel_i \mathit{Send}_i(d)$.

Because the specific typing of the read and send actions is not relevant, except that the function F_i must be compatible with it, we further consider a many-sorted setting. We assume that each variable is uniquely typed.

4.2 Connected Networks

A network is a finite collection of modules, in which the read/send connections respect the typing of the corresponding read and send actions. A general restriction is that there is *at most one* channel for transmission of between any two modules M_i and M_j. In particular, we do not allow merging of channels, or more than one feedback channel per module (case $i = j$). Note that branching of channels is modeled by taking different send actions. In this section we consider *connected* network specifications of the form

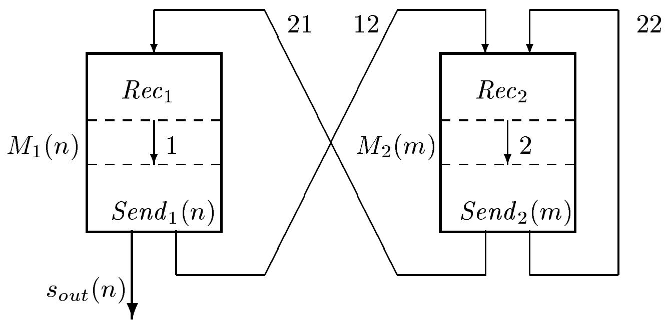
$$\tau_I \circ \partial_H \left(\left[\mathop{\parallel}_{i=1}^{n} M_i(d_i) \right] \right),$$

i.e., the graph of which the nodes are modules, and of which the (undirected) arcs are channels, is connected. In the specification above, the ∂_H application models value-passing synchronizations between modules $M_1, ..., M_n$, and the τ_I application models hiding of the resulting communication actions. We further say that the *I/O* of a network denotes its *external* actions, i.e., read or send actions that have no communication partner within the network. We call A network is *single-output* if its I/O consists of exactly one output action, which will be referred to as

$$s_{out}(...).$$

Below we recall an example taken from [BHP97] for computing a Fibonacci sequence using a connected single-output network consisting of modules M_1 and M_2. This example also illustrates the particular way we deal with *feedback*.

Example 4.2.1. Consider the following network in which all values to be passed are of type $\mathbb{N}$, and in which a channel name ij indicates that values are transmitted from module M_i to module M_j:



We can specify these modules by the following two iterative processes $M_1(n)$ and $M_2(m)$:

$$\begin{array}{rcl} M_1(n) & = & Rec_1 \parallel_1 Send_1(n), \\ Rec_1 & = & \big(er_{21}(v); s_1(v)\big)^\omega, \\ Send_1(n) & = & (s_{out}(n) \parallel s_{12}(n)) \cdot Send_1, \\ Send_1 & = & \big(er_1(v); (s_{out}(v) \parallel s_{12}(v))\big)^\omega, \\ \\ M_2(m) & = & Rec_2 \parallel_2 Send_2(m), \\ Rec_2 & = & \big((er_{12}(v_1) \parallel er_{22}(v_2)); s_2(v_1+v_2)\big)^\omega, \\ Send_2(m) & = & (s_{21}(m) \parallel s_{22}(m)) \cdot Send_2, \\ Send_2 & = & \big(er_2(v); (s_{21}(v) \parallel s_{22}(v))\big)^\omega. \end{array}$$

Let $I = \{c_{21}, c_{12}, c_{22}\}$ and $H = \{r_{21}, s_{21}, r_{12}, s_{12}, r_{22}, s_{22}\}$. The *Fibonacci Network*

$$\tau_I \circ \partial_H(M_1(1) \parallel M_2(1))$$

computes the ordinary Fibonacci sequence $1, 1, 2, 3, 5, 8, \ldots$ as the values of its consecutive s_{out}-actions:

$$\tau \cdot \tau_I \circ \partial_H(M_1(1) \parallel M_2(1)) = \tau \cdot s_{out}(1) \cdot s_{out}(1) \cdot s_{out}(2) \cdot s_{out}(3) \cdot s_{out}(5) \cdot s_{out}(8) \cdot \ \cdots$$

where the leftmost τ's smooth the difference between the networks first possible actions: either $s_{out}(1)$ or τ resulting from some internal value-passing. A different characterization is given by the equation

$$\tau \cdot \tau_I \circ \partial_H(M_1(n) \parallel M_2(m)) = \tau \cdot s_{out}(n) \cdot \tau_I \circ \partial_H(M_1(m) \parallel M_2(n+m)),$$

from which it is immediately clear that $\tau \cdot \tau_I \circ \partial_H(M_1(1) \parallel M_2(1))$ computes the Fibonacci sequence. This equation can easily be grasped from the picture above; its correctness follows from Theorem 4.3.1 presented in the following section.
End example.

4.3 Characterization of Connected, Single-I/O Networks

In this section two correctness results on connected single-I/O networks are recalled ([BHP97]), the second of which is a generalization of the first. We show by an example how a simple case of the first result can be proved.

Theorem 4.3.1. *Let $n \geq 1$, $\vec{d} = d_1, \ldots, d_n$ be a sequence of typed values, and let*

$$N(\vec{d}) = \tau_I \circ \partial_H \left(\left[\mathop{\parallel}_{i=1}^{n} M_i(d_i) \right] \right)$$

be a network that is connected and single-output, where M_1 is the output-module. Then

$$\tau \cdot N(\vec{d}) = \tau \cdot s_{out}(d_1) \cdot N(F_1(\vec{d_1}), \ldots, F_n(\vec{d_n}))$$

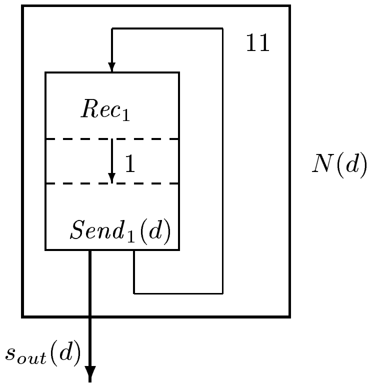
where F_i is the value function of module M_i, and $\vec{d_i}$ abbreviates $d_{i_1}, \ldots, d_{i_k}$ whenever F_i computes on the values of modules $M_{i_1}, \ldots, M_{i_k}$, respectively.

For the case $n = 1$, the proof of the theorem is trivial. We spell out a most simple instance. This also shows how a derivable variant of (2)—i.e., a data-parametric definition of a module with feedback—would look like.

Example 4.3.2. Consider the following network $N(d)$ where $d \in \mathbb{N}$, containing one module

$$M_1(d) = Rec_1 \parallel_1 Send_1(d)$$

that generates a stream:



Here

$$\begin{aligned}
Rec_1 &= \textstyle\sum(v : \mathbb{N},\ r_{11}(v) \cdot s_1(v+1))^\omega, \\
Send_1 &= \textstyle\sum(v : \mathbb{N},\ r_1(v)(s_{11}(v) \parallel s_{out}(v)))^\omega, \\
Send_1(d) &= (s_{11}(d) \parallel s_{out}(d)) \cdot Send_1.
\end{aligned}$$

Let $H = \{r_{11}, s_{11}\}$. The behaviour of $\partial_H(Rec_1 \parallel_1 Send_1(d))$ can be analyzed as follows:

$$\begin{aligned}
\partial_H(M_1(d)) &= \partial_H(Rec_1 \parallel_1 Send_1(d)) \\
&= c_{11}(d) \cdot \partial_H(s_1(d+1) \cdot Rec_1 \parallel_1 s_{out}(d) \cdot Send_1) \\
&\quad + s_{out}(d) \cdot \partial_H(Rec_1 \parallel_1 s_{11}(d) \cdot Send_1) \\
&= c_{11}(d) \cdot s_{out}(d) \cdot \partial_H(s_1(d+1) \cdot Rec_1 \parallel_1 Send_1) \\
&\quad + s_{out}(d) \cdot c_{11}(d) \cdot \partial_H(s_1(d+1) \cdot Rec_1 \parallel_1 Send_1) \\
&= c_{11}(d) \cdot s_{out}(d) \cdot c_1(d+1) \cdot \partial_H(Rec_1 \parallel_1 Send_1(d+1)) \\
&\quad + s_{out}(d) \cdot c_{11}(d) \cdot c_1(d+1) \cdot \partial_H(Rec_1 \parallel_1 Send_1(d+1)).
\end{aligned}$$

Let $I = \{c_{11}\}$, then it follows from the derivation above that the one-module network

$$N(d) = \tau_I \circ \partial_H(M_1(d))$$

for some $d \in \mathbb{N}$ satisfies

$$N(d) = \tau \cdot s_{out}(d) \cdot N(d+1) + s_{out}(d) \cdot N(d+1).$$

Hence

$$\tau \cdot N(d) = \tau \cdot s_{out}(d) \cdot N(d+1),$$

expressing that $\tau \cdot N(d)$ outputs the infinite stream $\tau \cdot s_{out}(d) \cdot s_{out}(d+1) \cdot s_{out}(d+2) \cdot \ \cdots$.
End example.

In general, a connected, single-output network with more than one module, all modules but the output module can be partitioned in a number of connected sub-networks that perform I/O with the output module only. From this perspective, the correctness Theorem 4.3.1 can be easily proved. The reader interested in a proof is referred to [BHP97].[1]

We can relax the conditions of Theorem 4.3.1 under which the execution of a network satisfies a single process prefix, followed by a recursive update of its data state. A first generalization concerns the output actions of a connected, single-output network. It is not hard to see that the previous correctness result is preserved if such a network outputs actions of the form

$$s_{out}(F(d))$$

for some function F rather than $s_{out}(d)$. We call this *output modification* of the *out*-channel.

A second generalization concerns *additional* external output of the network. Assume a network

$$N(\vec{d}) = \tau_I \circ \partial_H \left(\left[\mathop{\|}_{i=1}^{n} M_i(d_i) \right] \right)$$

has more than one output channel, and that I is such that all extra output channels not located at the I/O module are hidden. Then $N(\vec{d})$ is *single*-I/O if all its I/O activity (its collection of external read and send actions) stems from a single module, the I/O-*module*. Our most general result on the class of connected network is the following (cf. [BHP97], though we can exploit the Expansion Theorem and alphabet axioms, both recalled in Section 2, to obtain a nicer formulation):

Theorem 4.3.3. *Let $n \geq 1$, $\vec{d} = d_1, ..., d_n$ be a sequence of typed values, and let*

$$N(\vec{d}) = \tau_I \circ \partial_H \left(\left[\mathop{\|}_{i=1}^{n} M_i(d_i) \right] \right)$$

be a network that is connected and single-I/O, where M_1 is the I/O-module and $Extern$ is the set of indices of the I/O channels. (Notice that $Extern \neq \emptyset$ and may hide output from non-I/O-modules.)

Then

$$\tau \cdot N(\vec{d}) = \tau \cdot \left[\mathop{\|}_{i \in Extern} a_i(x_i) \right] \; ; \; \tau_I \circ \partial_H \left(\left[\mathop{\|}_{i=1}^{n} M_i(F_i(\vec{d_i})) \right] \right)$$

where

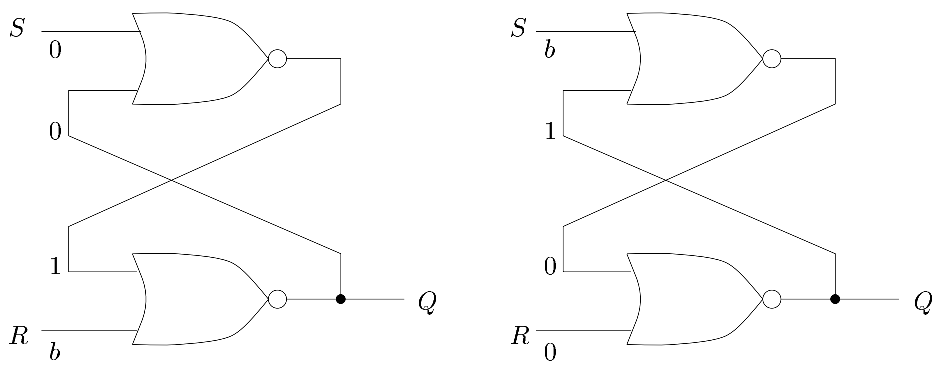
1. *Function F_i is the value function of module M_i,*
2. *For $i > 1$, $\vec{d_i}$ abbreviates $d_{i_1}, ..., d_{i_k}$ whenever F_i computes on the values of modules $M_{i_1}, ..., M_{i_k}$, respectively,*
3. *For $i \in Extern$, either $a_i(x_i) \equiv s_i(G_i(d_1))$ where G_i is the output modification of channel i, or $a_i(x_i) \equiv er_i(v_i)$,*
4. *Sequence $\vec{x_1}$ is defined similar, except for its $Extern$-coordinates (see the clause 3).*

This result gives way to regarding networks as *stream transformers*, be it that the I/O connection is located at a single module. In particular, this allows one to connect single-I/O networks with each other while preserving a simple correctness characterization.

[1] We remark that the last condition in the definition of Rec_l, i.e., $m \in R_l \setminus R' \Rightarrow x_m \in D_{j_m}$, should be skipped.

5 Beating Grid Protocols

In some cases the restriction to single-I/O networks is too strong. If for example one wants to model the operation of a simple *SR*-latch in process algebra (or *RS* Flip-Flop, cf. Section 5.1.2 in [TT94]), we obtain a network with multi-I/O. Below we depict an *SR*-latch in two typical states: irrespective of the value b, output at Q is 0, respectively 1. The components below symbolize *nor*-ports (where $nor(x, y) = 1 - max(x, y)$).



The format for connected networks discussed in the previous section does not comply to the intended operation of this network: it does not imply that S is evaluated before R starts its second evaluation. In this section we obtain the following type of characterization for *multi-I/O networks*, expressing that the I/O is performed *per cycle*:

$$\tau \cdot N(\vec{d}) = \tau \cdot ((\| \text{ all I/O value-passing actions}) \ ; \ N(\vec{F_i}(\vec{d_i})))$$

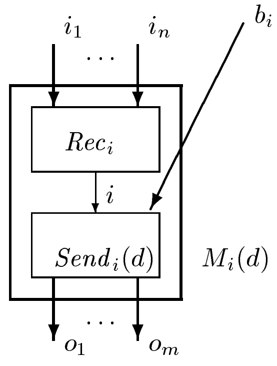
i.e., we want to view network operation in a similar way as module operation, performing I/O in consecutive phases. A way to achieve this is to assume a global synchronization device, the *Beat* process. We consider two different options for the definition of such a device.

5.1 Synchronized Modules

We follow the approach described in Section 4.1, but extend module specification with two external synchronization points with *Beat*, and one extra internal synchronization point. Another difference is that we associate a unique variable with *each* input channel:

$$\begin{aligned} M_i(d) &= Rec_i \parallel_i Send_i(d), \\ Rec_i &= \left(r_i\left(\left[\mathop{\|}_{j=1}^{n} er_{i_j}(v_{i_j})\right]; s_i(F_i(v_{i_1}, ..., v_{i_n}))\right)\right)^{\omega} \\ Send_i(d) &= r_{b_i}(s) \cdot s_i \cdot \left[\mathop{\|}_{j=1}^{m} s_{o_j}(d)\right] \cdot Send_i, \\ Send_i &= \left(er_i(v_i); \left(r_{b_i}(e) \cdot r_{b_i}(s) \cdot s_i \cdot \left[\mathop{\|}_{j=1}^{m} s_{o_j}(v_i)\right]\right)\right)^{\omega}. \end{aligned}$$

,



The idea is that $M_i(d)$ starts with a *Beat*-communication $c_{b_i}(s)$, whereupon Rec_i and $Send_i(d)$ synchronize with a c_i-communication and can start their parallel input and output actions. After this, Rec_i and $Send_i$ synchronize by value-passing action $c_i(F_i(\vec{d_i}))$, and "end-synchronization" with *Beat* can take place by a $c_{b_i}(e)$ communication, by which the module evolves into $M_i(F_i(\vec{d_i}))$. In the following section we further explain the synchronization actions between M_i and *Beat*.

5.2 Networks with a Beat

Again a network is a collection of modules as defined above, in which the read/send connections respect the typing of the read and send actions. As in the previous section we adopt the restriction that there is *at most one* channel for transmission of data from a module to a module (but feedback is allowed). For an n-module network, we consider two different definitions for the *Beat* process:

1. $Beat = \left(\left[\overset{n}{\underset{i=1}{\|}} s_{b_i}(s)\right] \cdot \left[\overset{n}{\underset{i=1}{\|}} s_{b_i}(e)\right]\right)^{\omega}$,
2. $Beat = \left(\left[\overset{n}{\underset{i=1}{\|}} s_{b_i}(s) \cdot s_{b_i}(e)\right]\right)^{\omega}$,

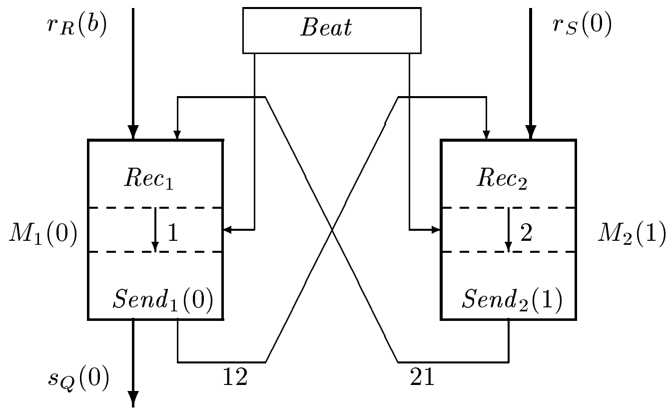
where the action $s_{b_i}(s)$ gives module M_i permission to start its parallel I/O, and $s_{b_i}(e)$ signals end of this activity. By definition of send-read communication, these actions yield communications $c_{b_i}(s)$ and $c_{b_i}(e)$. Note that the second definition of *Beat* is most liberal, as it covers each execution of the first one. (The converse is not true: *all* $s_{b_j}(s)$ actions must take place before an $s_{b_i}(e)$ action can occur.) In this section we consider network specifications of the form

$$\tau_I \circ \partial_H \left(\left[\overset{n}{\underset{i=1}{\|}} M_i(d_i)\right] \| \; Beat\right),$$

in which the ∂_H application models value-passing synchronizations between modules $M_1, ..., M_n$ and *Beat*, and the τ_I application models hiding of the resulting communication actions. In this case, the rhythm of the *Beat* guides the operation of the network.

Below we give an example for computing the operation of an *SR*-latch using a beating grid protocol.

Example 5.2.1. *SR-Latch.* Consider the following network in which all data to be transmitted are in $\{0,1\}$. A channel name ij indicates that values are transmitted from module M_i to module M_j: We put the branching of output of the R, Q-module explicit in the module (corresponding with the restriction that channels have bandwidth 1):



The precise specification of $M_1(b_1)$ and $M_2(b_2)$ is as follows:

$$\begin{aligned}
M_1(b_1) &= Rec_1 \parallel_1 Send_1(b_1), \\
Rec_1 &= (r_1 \cdot [(er_R(v_R) \parallel er_{21}(v_{21})); s_1(nor(v_R, v_{21}))])^\omega, \\
Send_1(b_1) &= r_{b_1}(s) \cdot s_1 \cdot (s_Q(b_1) \parallel s_{12}(b_1)) \cdot Send_1, \\
Send_1 &= (er_1(v); [r_{b_1}(e) \cdot r_{b_1}(s) \cdot s_1 \cdot (s_Q(v) \parallel s_{12}(v))])^\omega, \\
M_2(b_2) &= Rec_2 \parallel_2 Send_2(b_2), \\
Rec_2 &= (r_1 \cdot [(er_S(v_S) \parallel er_{12}(v_{12})); s_2(nor(v_S, v_{12}))])^\omega, \\
Send_2(b_2) &= r_{b_2}(s) \cdot s_2 \cdot s_{21}(b_2) \cdot Send_2, \\
Send_2 &= (er_2(v); [r_{b_2}(e) \cdot r_{b_2}(s) \cdot s_2 \cdot s_{21}(v)])^\omega.
\end{aligned}$$

Let $I = \{c_{21}, c_{12}\}$ and $H = \{r_{21}, s_{21}, r_{12}, s_{12}\}$. We argue that

$$\tau_I \circ \partial_H(M_1(0) \parallel M_2(1) \parallel Beat)$$

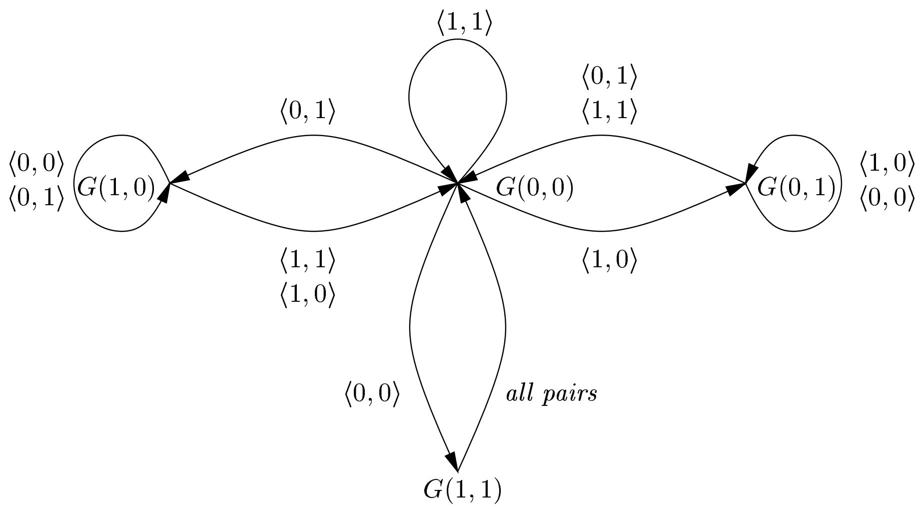
computes the operation of an *SR*-latch per *two* cycles. First we analyze the behaviour of

$$G(b_1, b_2) \stackrel{\text{def}}{=} \tau_I \circ \partial_H(M_1(b_1) \parallel M_2(b_2))$$

in a graphical style. The characterization theorem for beating grid protocols, which we present below, yields that

$$\tau \cdot G(b_1, b_2) = \tau \cdot ((er_R(c_1) \parallel er_S(c_2) \parallel s_Q(b_1)); G(nor(b_2, c_1), nor(b_1, c_2))).$$

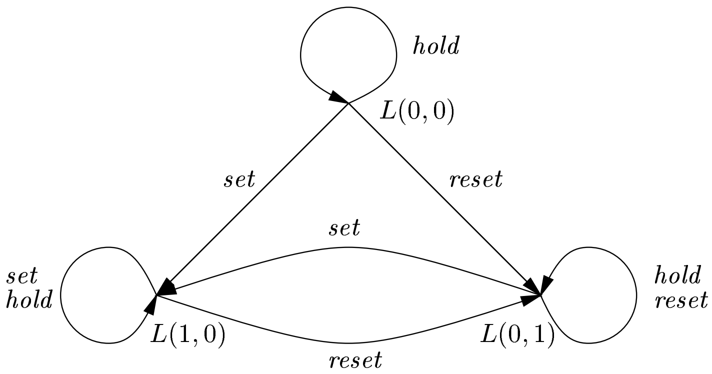
In order to further analyze this behaviour we use a graphical style, deleting τ-steps and only showing the input-value pairs $\langle c_R, c_S \rangle$. The output value b_1 of $s_Q(b_1)$ is characterized as the first value of $G(\_, \_)$:



As argued in [TT94], the *SR*-latch behaviour can be traced back if one assumes that input is offered twice per cycle to the network. Let $L(b_1, b_2)$ represent the appropriate G-states, and use the following coding of input pairs:

set	for	$\langle 0,1\rangle \cdot \langle 0,1\rangle$	(output 1 at Q)
reset	for	$\langle 1,0\rangle \cdot \langle 1,0\rangle$	(output 0 at Q)
hold	for	$\langle 0,0\rangle \cdot \langle 0,0\rangle$	(output at Q the same as in the previous cycle)

Then $L(0,0)$ has the intended behaviour, as follows from the behaviour of $G(0,0)$ as analyzed above:



End example.

5.3 Characterization of Beating Grid Protocols

In this section we state our final correctness results on networks. As before, we allow *output modification* of the external output-actions.

Theorem 5.3.1. *Let $n \geq 1$, $\vec{d} = d_1, ..., d_n$ be a sequence of typed values, and let*

$$N(\vec{d}) = \tau_I \circ \partial_H \left(\left[\mathop{\|}_{i=1}^{n} M_i(d_i) \right] \ \| \ Beat \right)$$

be a beating grid protocol, with synchronized modules. Furthermore, let Beat be defined in one of the following ways:

$$1. \quad Beat = \left(\left[\mathop{\|}_{i=1}^{n} s_{b_i}(s) \right] \cdot \left[\mathop{\|}_{i=1}^{n} s_{b_i}(e) \right] \right)^{\omega},$$

$$2. \quad Beat = \left(\left[\mathop{\|}_{i=1}^{n} s_{b_i}(s) \cdot s_{b_i}(e) \right] \right)^{\omega}.$$

Then

$$\tau \cdot N(\vec{d}) = \tau \cdot \left[\mathop{\|}_{i \in Extern} a_i(x_i) \right] \ ; \ \tau_I \circ \partial_H \left(\left[\mathop{\|}_{i=1}^{n} M_i(F_i(\vec{d_i})) \right] \| \ Beat \right),$$

where

1. *$Extern \neq \emptyset$ is the set of indices of the I/O channels,*
2. *Function F_i is the value function of module M_i,*

3. *For $i \in Extern$, either $a_i(x_i) \equiv s_i(G_i(d_i))$ where G_i is the output modification of channel i, or $a_i(x_i) \equiv er_i(v_i)$,*

4. *The sequence $\vec{d_i}$ abbreviates $a_{i_1}, ..., a_{i_{k_i}}$ whenever F_i computes on input channels $i_1, ..., i_{k_i}$.*

Various inductive proofs of this result [BJM97, Pou97] use a second characterization that covers the case that a network has *no* external connections.

6 An Example: the Wave Equation

In this section we specify a given algorithm for approximation of a wave equation in a single-output and connected network. The description of this example is taken from [BHP97].

6.1 The wave equation

The linear homogeneous partial differential equation

$$\frac{\partial^2 y}{\partial t^2} - c^2 \frac{\partial^2 y}{\partial x^2} = 0$$

is known in wave mechanics as the *one-dimensional wave equation*; it describes transversal propagation along the x-coordinate or *amplitude* $y(x,t)$ of a wave. This equation models for instance vibrations in a string, where it is required that the tension in the string is approximately constant. The constant c is defined by $\sqrt{T/\rho}$, where T is the tension in the string and ρ the string mass per unit of length. In solutions $y(x,t)$ the constant c is interpreted as the propagation velocity of the wave in transversal direction.

In order to solve the wave equation, boundary and initial conditions are needed. As boundary conditions we assume that $y(0,t) = y(l,t) = 0$ for $t \geq 0$, i.e. that the string is fixed in $x = 0$ and $x = l$. With these boundary conditions a string amplitude at some time t, as a function of x, may be graphically represented as in Fig. 2.

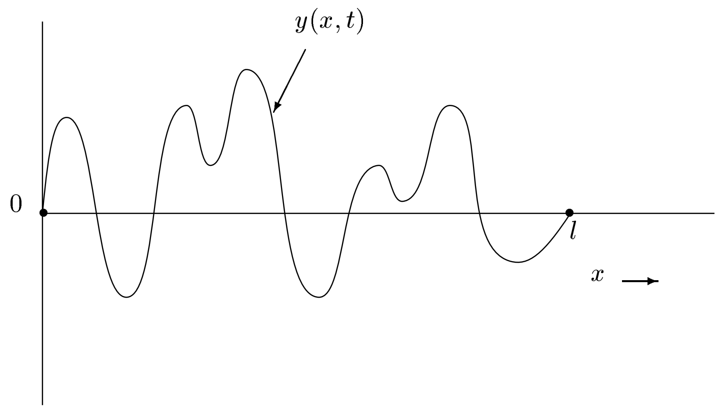


Figure 2: A string amplitude at time t

In case we have $y(x,0)$ and $\partial y/\partial t|_{t=0}$ given as initial conditions for $0 \leq x \leq l$, it is possible to derive an approximation of $y(x, \Delta t)$, where Δt is a very small time interval. The values $y(x,0)$ and $y(x,\Delta t)$ are used for the initialization of an algorithm that numerically solves the wave equation.

Let N be a natural number, and $\Delta x = l/N$ a very small length interval. We define

$$F(z_1, z_2, z_3, z_4) = 2z_1 - z_2 + (c\frac{\Delta t}{\Delta x})^2(z_3 - 2z_1 + z_4),$$

and

$$y_i(t + \Delta t) = F(y_i(t), y_i(t - \Delta t), y_{i-1}(t), y_{i+1}(t)).$$

From numerical analysis it is known that $y_i(t)$ approximates $y(i\Delta x, t)$ for $1 \leq i \leq N - 1$, and $t \geq 2\Delta t$ (see e.g. [Smi65, FJL$^+$88]). Therefore, the above equation for $y_i(t+\Delta t)$ may serve as a basis for numerical approximation of solutions of the wave equation.

Now an algorithm for calculating wave amplitudes $y_i(t)$ may be designed which uses one processor per *sample point* on the x-axis, i.e., one for every i and one for each boundary. As a result the calculations for the string amplitude at some *sample moment* t will be carried out by $N+1$ processors in parallel. In fact, $N-1$ processors will suffice, since the values at the sample points $i = 0$ and $i = N$ are already known from the boundary conditions.

In the next section we specify a connected, single-output grid protocol that models this approximation. For simplicity we assume that Δx and Δt are given, and that there is no interaction between a user of the algorithm and the algorithm itself; the algorithm just produces an infinite stream of outputs. Of course we need a criterion for correctness; we require that the algorithm outputs approximations of the total string amplitudes on the successive sample moments:

$$\vec{y}(0), \vec{y}(\Delta t), \vec{y}(2\Delta t), \ldots,$$

where $\vec{y}(t)$ abbreviates $y_0(t), \ldots, y_N(t)$. Other requirements are that the algorithm contains no deadlocks or livelocks, so that it is always able to proceed. We will see from one simple equation on the external behaviour of the algorithm that these three requirements are satisfied. This equation immediately follows from the correctness theorems presented earlier.

6.2 Grid Protocols modeling the Wave

In the previous section we established the following equation for the calculation of the value of coordinate y_i at time $t + \Delta t$:

$$y_i(t + \Delta t) = F(y_i(t), y_i(t - \Delta t), y_{i-1}(t), y_{i+1}(t)).$$

Modeling this approximation as a grid protocol obscures explicit reference to *time*. All that is left is that computation of the amplitudes at time $t + \Delta t$ depends on the amplitudes at time t and time $t - \Delta t$, and that our protocol is modeled in such a way that *all* outputted amplitudes are computed in consecutive phases. So any reference to time t must be interpreted as to a certain computation phase.

The equation above shows that the current values (at time t) of coordinates y_i, y_{i-1}, and y_{i+1} are needed, as well as the previous value (at time $t - \Delta t$) of coordinate y_i. Given these values, the function F calculates the new value (at time $t + \Delta t$) of y_i. When we model the approximation of the wave equation as a grid protocol, we need a number of processors, each calculating the consecutive values of one or more coordinates as floating reals. We choose to let one processor calculate the values of one coordinate. For $N + 1$ coordinates, we thus define $N + 1$ processors $P_0 \ldots P_N$. Each processor P_i $(0 < i < N)$ needs the following input:

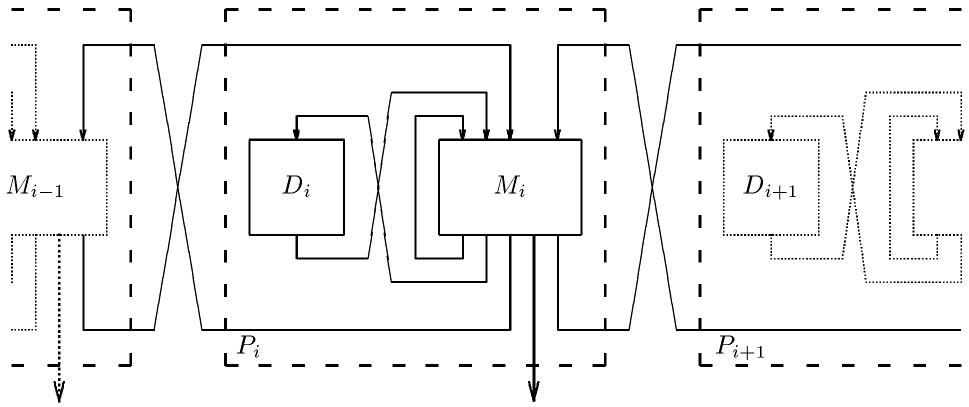
- the output of processor P_{i-1},

- the output of processor P_i (itself),
- the output of processor P_{i+1}, and
- the *previous* output of processor P_i (itself).

Naturally, processors P_0 and P_N do not need input at all. However, for reasons of uniformity we also use channels from P_1 to P_0 and from P_{N-1} to P_N. The last item above requires that we store the output of each processor for one time slot. This is, however, not possible in a single module. We solve this problem by splitting each processor P_i into a calculating module M_i and a delay module D_i. The delay module does nothing more than storing the output value of the calculating module for one time slot. After that, this value is sent back to the calculating module. We can now state that the input of each module M_i $(0 < i < N)$ should be:

- the output of module M_{i-1},
- the output of module M_i (itself),
- the output of module M_{i+1}, and
- the output of module D_i.

We can visualize this as follows:



Now we can proceed in two ways:

1. We can define a *connected*, single-output grid protocol. Here an additional output module transmits the values of all 'processors', and hence acts as a synchronizing device,
2. We can define a single-output *beating* grid protocol, collecting output directly from the 'processors' described above.

We only describe the first alternative. It should be immediately clear how to model the second approach. Now that we have composed a grid protocol modeling the wave equation, we can start writing a specification in the early-read format. This is not difficult: just read what happens from the picture. To start with, we specify the D_i $(0 < i < N)$ modules:

$$\begin{aligned}
D_i(e) &= \tau_{\{c_{(D_i)}\}} \circ \partial_{\{r_{(D_i)}, s_{(D_i)}\}}(RD_i \parallel SD_i(e)) \\
RD_i &= (er_{(M_i,D_i)}(v); s_{(D_i)}(v))^\omega \\
SD_i &= (er_{(D_i)}(v); s_{(D_i,M_i)}(v))^\omega \\
SD_i(e) &= s_{(D_i,M_i)}(e) \cdot SD_i.
\end{aligned}$$

Here, $er_{(M_i,D_i)}(v)$ and $s_{(D_i,M_i)}(v)$ stand for an early read or a send action on the ports connecting M_i and D_i. Note that (M_i, D_i) is the port from M_i to D_i and (D_i, M_i) the port from D_i to M_i. The actions $er_{(D_i)}(v)$ and $s_{(D_i)}(v)$ stand for an early read or a send action on the internal port of the concerning module.

Likewise, we specify the modules M_i:

$$\begin{array}{rcl}
M_i(d) & = & \tau_{\{c_{(M_i)}\}} \circ \partial_{\{r_{(M_i)}, s_{(M_i)}\}}(R_i \parallel S_i(d)) \\
R_i & = & \big((er_{(M_i,M_i)}(v_1) \parallel er_{(D_i,M_i)}(v_2) \parallel er_{(M_{i-1},M_i)}(v_3) \parallel er_{(M_{i+1},M_i)}(v_4))\ ; \\
 & & s_{(M_i)}(F(v_1,v_2,v_3,v_4))\big)^\omega \\
S_i & = & (er_{(M_i)}(v); (s_{(M_i,M_i)}(v) \parallel s_{(M_i,D_i)}(v) \parallel s_{(M_i,M_{i-1})}(v) \parallel s_{(M_i,M_{i+1})}(v) \parallel s_{(M_i,O)}(v)))^\omega \\
S_i(d) & = & (s_{(M_i,M_i)}(d) \parallel s_{(M_i,D_i)}(d) \parallel s_{(M_i,M_{i-1})}(d) \parallel s_{(M_i,M_{i+1})}(d) \parallel s_{(M_i,O)}(d)) \cdot S_i.
\end{array}$$

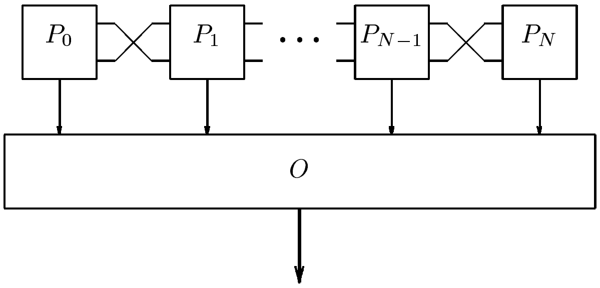
The port (M_i, O) is the actual output port of the processor, leading to an output module O.

The processors P_i $(0 < i < N)$ can now be defined as follows:

$$P_i(d,e) = M_i(d) \parallel D_i(e),$$

with d and e the initial values of coordinate y_i ($y_i(\Delta t)$ and $y_i(0)$, respectively).

For $N+1$ coordinate pairs, $N-1$ of these processors are coupled together, the outer ones also using two border processors (which are simple modules). The output of all the calculating modules M_i $(0 \leq i \leq N)$ in the processors is sent to output module O. This module collects the computed values of all processors and bundles them in a vector. This bundling is somewhat arbitrary; alternatively O may send its output in parallel to the environment. In a picture:



As one can see from this picture, the first and the last processor only communicate with their neighbor and the output module O. The specification of these two processors is, therefore, very simple:

$$\begin{array}{rcl}
P_0(d) & = & \tau_{\{c_{(P_0)}\}} \circ \partial_{\{r_{(P_0)}, s_{(P_0)}\}}(R_0 \parallel S_0(d)) \\
R_0 & = & (er_{(M_1,P_0)}(v); s_{(P_0)}(0))^\omega \\
S_0 & = & (er_{(P_0)}(v); (s_{(P_0,M_1)}(v) \parallel s_{(P_0,O)}(v)))^\omega \\
S_0(d) & = & (s_{(P_0,M_1)}(d) \parallel s_{(P_0,O)}(d)) \cdot S_0
\end{array}$$

$$\begin{array}{rcl}
P_N(d) & = & \tau_{\{c_{(P_N)}\}} \circ \partial_{\{r_{(P_N)}, s_{(P_N)}\}}(R_N \parallel S_N(d)) \\
R_N & = & (er_{(M_{N-1},P_N)}(v); s_{(P_N)}(0))^\omega \\
S_N & = & (er_{(P_N)}(v); (s_{(P_N,M_{N-1})}(v) \parallel s_{(P_N,O)}(v)))^\omega \\
S_N(d) & = & (s_{(P_N,M_{N-1})}(d) \parallel s_{(P_N,O)}(d)) \cdot S_N.
\end{array}$$

Note that P_0 and P_N need not to be split in a calculating and a delay module. Since we describe a wave through a string with both ends tight, the output value of processors P_0 and P_N will remain zero all the time:

$$P_0(d,e) = P_0(0) \quad \text{and} \quad P_N(d,e) = P_N(0).$$

The only thing left to specify is the output module O:

$$\begin{aligned}
O(d_0,\ldots,d_N) &= \tau_{\{c_{(O)}\}} \circ \partial_{\{r_{(O)},s_{(O)}\}}(RO \parallel SO(d_0,\ldots,d_N)) \\
RO &= \big((er_{(P_0,O)}(v_0) \parallel er_{(M_1,O)}(v_1) \parallel \ldots \parallel er_{(M_{N-1},O)}(v_{N-1}) \parallel er_{(P_N,O)}(v_N))\ ; \\
&\quad\ s_{(O)}(v_0,\ldots,v_N)\big)^{\omega} \\
SO &= (er_{(O)}(w_0,\ldots,w_N); s_{out}(w_0,\ldots,w_N))^{\omega} \\
SO(d_0,\ldots,d_N) &= s_{out}(d_0,\ldots,d_N) \cdot SO.
\end{aligned}$$

Now the algorithm is specified by the parallel composition of O and all processors P_i:

$$\text{WAVE} = \tau_{\{c_p\}} \circ \partial_{\{r_p,s_p\}} \left(O(\vec{y}(0)) \quad \Big\| \quad \left[\mathop{\Big\|}_{i=0}^{N} P_i(y_i(\Delta t), y_i(0)) \right] \right),$$

with $y_i(0), y_i(\Delta t)$ $(i = 0 \ldots N)$ arbitrary initial values, and p ranging over the following set of ports:

$$\begin{aligned}
&\{(M_i,M_j),(M_i,D_i),(D_i,M_i),(M_i,O) \mid 0 < i,j < N\} \ \cup \\
&\{(P_0,M_1),(M_1,P_0),(P_0,O),(M_{N-1},P_N),(P_N,M_{N-1}),(P_N,O)\}.
\end{aligned}$$

The external behaviour of the algorithm can then be expressed by

$$\tau \cdot \text{WAVE} = \tau \cdot s_{out}(\vec{y}(0)) \cdot s_{out}(\vec{y}(\Delta t)) \cdot s_{out}(\vec{y}(2\Delta t)) \cdot \ \cdots .$$

This follows from Theorem 4.3.1, which gives the following characterization of our specification:

$$\begin{aligned}
&\tau \cdot \tau_{\{c_p\}} \circ \partial_{\{r_p,s_p\}} \left(O(\vec{y}(k \cdot \Delta t)) \quad \Big\| \quad \left[\mathop{\Big\|}_{i=0}^{N} P_i(y_i((k+1) \cdot \Delta t), y_i(k \cdot \Delta t)) \right] \right) \quad = \\
&\tau \cdot s_{out}(\vec{y}(k \cdot \Delta t)) \cdot \\
&\quad \tau_{\{c_p\}} \circ \partial_{\{r_p,s_p\}} \left(O(\vec{y}((k+1) \cdot \Delta t)) \quad \Big\| \quad \left[\mathop{\Big\|}_{i=0}^{N} P_i(y_i((k+2) \cdot \Delta t), y_i((k+1) \cdot \Delta t)) \right] \right).
\end{aligned}$$

A beating grid protocol does not depend on an output module like O as defined above. For such a protocol one has a choice in the definition of its ouput values: either those of M_i, yielding

$$\tau \cdot \text{WAVE}' = \tau \cdot \left(\left[\mathop{\Big\|}_{i=0}^{N} s_{M_i,O}(y_i(\Delta t)) \right] \right) \cdot \left(\left[\mathop{\Big\|}_{i=0}^{N} s_{M_i,O}(y_i(2\Delta t)) \right] \right) \cdot \ldots$$

not giving output at time 0, or reading the values from the D_i modules instead. In this case, the modules M_i lose their output action $s_{M_i,O}(..)$, and the D_i should get an extra output action $s_{D_i,O}(..)$. We then obtain also the output at time 0.

7 Conclusions

It seems that early read and process prefix form a useful extension to ACP-based specification formalisms. Future work concerns a more formal treatment of substitution, extensions in the field of asynchronous networks, and establishing a precise relationship with protocol specification and verification in μCRL.

We finish with some remarks on the present text. First we remark on specification format: it is not essential whether one uses full μCRL, or the ω-extension of ACP with data as presented in this paper as basic specification language. Restricting all occurring types of networks to single-module networks, one finds by the characterization results that an identity such as

$$M(d) = \left[\underset{I/O}{\|}\ actions]\right] ; M(F(\vec{v}))$$

can also be regarded as the (data-parametric) *specification* of a module. In that case, transformation to the ω-extension brings one in a proof theoretic mode in which characterization seems relatively easy to prove. Note that the μCRL perspective yields two basic types of modules, both in the setting without and with a beat process: those *without* feedback, as displayed above, and those *with* feedback, having a value-update of the form $F(d, \vec{v})$ (and the possibility of an initial τ-step if one cares to model the feedback as explicit activity, which from an operational point of view seems best).

Furthermore, one can show that for the internal communication actions it is sufficient to assume that all outputs are in parallel; input may have a fixed order. Because all internal output is performed in parallel this cannot raise any deadlock, which also is a consequence of the characterization results: both approaches reduce to the same external characterization. This can be useful for proto-typing of grid protocols (cf. [Hil96, Pou97]).

References

[BB94] J.C.M. Baeten and J.A. Bergstra. On sequential composition, action prefixes and process prefix. *Journal of Formal Aspects of Computing Science*, 6(3):83–98, 1994.

[BBK87] 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, Proceedings of the* 3^{th} *IFIP WG 2.2 working conference,* Ebberup 1986, pages 53–75, Amsterdam, 1987. North-Holland.

[BHP97] J.A. Bergstra, J.A. Hillebrand, and A. Ponse. Grid protocols based on synchronous communication, *Science of Computer Programming*, 29:199-233, 1997.

[BJM97] E. van Buiten, M. de Jonge, and R. Monajemi. Beating Grid Protocols, PAII-thesis, University of Amsterdam, 1997.

[Pou97] M. Pouw. Beating Grid Protocols, Master's Thesis, Utrecht University, 1997.

[BK84] J.A. Bergstra and J.W. Klop. The algebra of recursively defined processes and the algebra of regular processes. In J. Paredaens, editor, *Proceedings* 11^{th} *ICALP,* Antwerpen, volume 172 of *Lecture Notes in Computer Science*, pages 82–95. Springer-Verlag, 1984. An extended version appeared in [PVV95], pages 1–25, 1995.

[BK85] J.A. Bergstra and J.W. Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science*, 37(1):77–121, 1985.

[BT84] J.A. Bergstra and J.V. Tucker. Top down design and the algebra of communicating processes. *Science of Computer Programming*, 5(2):171–199, 1984.

[BT95] J.A. Bergstra and J.V. Tucker. Equational specifications, complete term rewriting systems, and computable and semicomputable algebras. *Journal of the ACM*, 42(6):1194–1230, 1995.

[BW90] J.C.M. Baeten and W.P. Weijland. *Process Algebra.* Cambridge Tracts in Theoretical Computer Science 18. Cambridge University Press, 1990.

[FJL$^+$88] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *General Techniques and Regular Problems*, volume 1 of *Solving Problems on Concurrent Processors.* Prentice-Hall International, 1988.

[Fok97] W.J. Fokkink, Axiomatizations for the perpetual loop in process algebra. In P. Degano, R. Gorrieri, and A. Marchetti-Spaccamela, editors, *Proc. 24th Colloquium on Automata, Languages and Programming - ICALP'97*, pages 571–581. Lecture Notes in Computer Science Vol. 1256, Springer-Verlag, Berlin, 1997.

[GK95] J.F. Groote and H. Korver. A correctness proof of the bakery protocol in μCRL. In [PVV95], pages 63–86, 1995.

[GP91c] J.F. Groote and A. Ponse. Proof theory for μCRL. (Extended version.) Report CS-R9138, CWI, Amsterdam, 1991.

[GP94b] 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, *Proceedings of the International Workshop on Semantics of Specification Languages*, pages 232–251. Workshops in Computing, Springer-Verlag, 1994.

[GP95] J.F. Groote and A. Ponse. The syntax and semantics of μCRL. In [PVV95], pages 26–62, 1995.

[Hil96] J.A. Hillebrand. A simple language for the specification of grid protocols (working title). Technical Report, Programming Research Group, University of Amsterdam, to appear.

[Kle56] S.C. Kleene. Representation of events in nerve nets and finite automata. In *Automata Studies*, pages 3–41. Princeton University Press, 1956.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice-Hall International, Englewood Cliffs, 1989.

[PVV95] A. Ponse, C. Verhoef, and S.F.M. van Vlijmen, editors. *Algebra of Communicating Processes, Utrecht 1994*, Workshops in Computing. Springer-Verlag, 1995.

[Smi65] G.D. Smith. *Numerical Solution of Partial Differential Equations*, Oxford University Press, 1965.

[TT94] B.C. Thompson and J.V. Tucker. Equational specification of Synchronous Concurrent Algorithms and architectures (Second Edition). Report CSR 15-94, University of Wales, Swansea, 1994.