ELSEVIER Science of Computer Programming 29 (1997) 199–233

Science of Computer Programming

Grid protocols based on synchronous communication

Jan A. Bergstra[a,b], Joris A. Hillebrand[a], Alban Ponse[a,*]

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

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

Abstract

We provide a short notation for processes with parallel inputs and outputs. With this specification format synchronous networks or grid protocols can be specified in a straightforward way. For a certain class of connected networks we prove a correctness theorem that characterizes I/O behavior. We illustrate our approach by an example on the approximation of a one-dimensional wave equation. © 1997 Elsevier Science B.V.

Keywords: Grid protocol; Parallel computation; Stream transformers; Parallel input and output; Synchronous networks

1. Introduction

A *grid protocol* is a network that can be associated with parallel computation in a grid-like architecture. Such an architecture can be a network of processors or (groups of) points of measure in some physical phenomenon, for example a vibrating string. A grid protocol is assumed to consist of *modules*, elementary processors of data that can cooperate with each other by passing values. This cooperation can be modeled in various ways. In this paper we consider value-passing by synchronization (communication actions). A module is characterized by a predefined function (its computational identity), a current value, and a finite number of channels (or ports). A channel models the connection with either one of the network's modules, or with some external device. In terms of behavior, a module repeatedly performs the parallel execution of input and output actions (each one operating on a distinct channel), followed by an update of its current value. This value update results from application of the module's function to the newly received value(s). In the case that all modules and internal channels of the network form a connected graph and the external behavior is located at one module, we obtain a simple characterization result: the order of the (internal) synchronizations is not relevant and the network's external behavior – *stream* transformation or generation – is determined by simultaneous value updates.

* Corresponding author. E-mail: alban@fwi.uva.nl.

PII S0167-6423(96)00035-4

The point of departure is a combination of value-passing calculus CCS (Calculus of Communicating Systems [20]) and the process algebraic approach ACP (Algebra of Communicating Processes [3, 5, 6]). The technical construct underlying our approach is the *process prefix*, a generalization of Milner's action prefix which provides a means for binding variables in CCS. With the process prefix and so called *early read* actions, a concise notation of parallel input is possible. In [1], Baeten and Bergstra proposed axiom systems for action prefixes, process prefixes, and early read actions in the setting of ACP. In order to specify and analyze grid protocols we need to extend the process prefixing mechanism of [1] to an infinitary setting. For the specification of computable data and value-passing we use some machinery of μCRL [15], an ACP-based approach in which both data and processes can be formally specified and analyzed.

In terms of computation theory, our approach does not add to research performed elsewhere, e.g., concerning simultaneous primitive recursion theory [1]. We only provide results about a simple class of networks. A motivation for this work is to present an operational perspective on the module level – *value-passing by arbitrary interleaved synchronizations* – and to relate this perspective to a correctness characterization about a network's external input/output behavior.

After a brief introduction to the axiom system $\mathrm{ACP}^{\tau}(A,\gamma)$, iteration and alphabet axioms (Section 2), we present $\mathrm{ACP}^{\tau}_{er}(A,\gamma)$, which stands for $\mathrm{ACP}^{\tau}(A,\gamma)$ with process prefixes and early-read actions (Section 3). Then, in Section 4, we define *modules*. For finite, connected networks with output located at one port, we present in Section 5 a simple equation that characterizes external behavior, and hence *correctness* of the specification. In Section 6 we further generalize our correctness result to a type of networks that can consume input. We illustrate our specification format for grid protocols in Section 7 by a parallel algorithm for the numerical computation of solutions of the one-dimensional *wave equation*, which is a partial differential equation describing elementary wave phenomena, such as the transversal propagation of vibrations in a string. In a straightforward manner, the algorithm is specified as a connected network, from which its correctness follows. Furthermore, we pay some attention to the elimination of process prefixes and early read actions in the specification, and to simulation issues. Section 8, containing some conclusions, ends the paper.

1.1. Related work

Our modeling of modules and grid protocols is very much based on the work done on synchronous concurrent algorithms (SCAs) in Swansea [22]. A particular reason to follow the Swansea approach is given by the following citation: "many specialised

[1] In case all module functions are primitive recursive. Notice that in a many-sorted setting, this type of recursion is inequivalent to non-simultaneous primitive recursion over many-sorted structures (see [22] for further details and references).

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". We think that our example on the wave equation supports this claim, and that many practical examples can be obtained from the work done on SCAs.

As mentioned above, in [1] the authors develop a different approach to process prefixing. They circumvent the use of typed *variables* in value-passing, and related questions of bound variables and α-conversion. A price to be paid is the restriction to a *finite* data type. In [7] a more abstract approach is followed (network algebra for synchronous and asynchronous dataflow).

Finally, in [16] a tool is described that translates a specification in the early-read format into a standard μCRL specification (without early reads). This gives way to simulation tools and standard μCRL proof theory.

2. Process algebra, 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 iteration. We quote an expressivity result on $\mathrm{ACP}^\tau(A,\gamma)$ with iteration, and give a new, short proof. Finally, we discuss generalized merges, expansion and alphabet axioms, all of which are essential for the specification and verification of grid protocols.

2.1. $ACP^\tau(A,\gamma)$, standard concurrency and 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 are associative.

We further assume commutativity and associativity of $\|$ and $|$, also known as SC (standard concurrency [8]). In this paper we only consider two-party communication or *handshaking*, axiomatized by $x \mid y \mid z = \delta$ (see [8]). For a detailed introduction to $\mathrm{ACP}^\tau(A,\gamma)$ and SC we refer to [3].

Table 1
The axioms of $\mathrm{ACP}^{\tau}(A,\gamma)$ and for the binary Kleene star, 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)	δx	$= \delta$			
(CF1)	$a \mid b$	$= \gamma(a,b)$ if $\gamma(a,b)\downarrow$			
(CF2)	$a \mid b$	$= \delta$ otherwise			
(CM1)	$x \parallel y$	$= x \between y + y \between x + x \mid y$	(D1)	$\partial_H(a)$	$= a$ if $a \notin H$
(CM2)	$a \between x$	$= ax$	(D2)	$\partial_H(a)$	$= \delta$ if $a \in H$
(CM3)	$ax \between y$	$= a(x \parallel y)$	(D3)	$\partial_H(x+y)$	$= \partial_H(x)+\partial_H(y)$
(CM4)	$(x+y) \between z$	$= x \between z + y \between z$	(D4)	$\partial_H(xy)$	$= \partial_H(x)\cdot\partial_H(y)$
(CM5)	$ax \mid b$	$= (a \mid b)x$			
(CM6)	$a \mid bx$	$= (a \mid b)x$	(TI1)	$\tau_I(a)$	$= a$ if $a \notin I$
(CM7)	$ax \mid by$	$= (a \mid b)(x \parallel y)$	(TI2)	$\tau_I(a)$	$= \tau$ if $a \in I$
(CM8)	$(x+y) \mid z$	$= x \mid z + y \mid z$	(TI3)	$\tau_I(x+y)$	$= \tau_I(x)+\tau_I(y)$
(CM9)	$x \mid (y+z)$	$= x \mid y + x \mid z$	(TI4)	$\tau_I(xy)$	$= \tau_I(x)\cdot\tau_I(y)$
(BKS1)	x^*y	$= x(x^*y)+y$	(BKS4)	$\partial_H(x^*y)$	$= \partial_H(x)^*\partial_H(y)$
(BKS2)	$x^*(yz)$	$= (x^*y)z$	(BKS5)	$\tau_I(x^*y)$	$= \tau_I(x)^*\tau_I(y)$
(BKS3)	$(x+y)^*z$	$= x^*(y((x+y)^*z)+z)$			

In order to describe iterative processes we shall use the (binary) Kleene star [4, 17], of which the defining axiom is

$$\text{(BKS1)} \quad x^*y = x(x^*y)+y.$$

So x^*y is the process that chooses between x and y, and upon termination of x has this choice again. Thus, if x is a terminating process then $x^*\delta$ is the process that repeatedly executes x. Remaining axioms for the *-operation are included in Table 1. In [10], Fokkink and Zantema prove that A1–A5 and BKS1–BKS3 axiomatize strong bisimilarity for processes defined with $+, \cdot$ and *.

For the interested reader, we elaborate a little on the way one can reason with iterative processes. An advantage of the *-operation is that one can reason equationally on infinite processes. As a trivial example, consider

$$\partial_{\{b\}}(a^*b) \overset{\text{BKS4}}{=} \partial_{\{b\}}(a)^*\partial_{\{b\}}(b) \overset{\text{D1,2}}{=} a^*\delta.$$

Finally, we quote the following expressivity result on regular processes,[2] and give a new, short proof.

[2] That is, processes specifiable by a finite, linear system of recursive equations.

Theorem 2.1 ([4, Theorem 3.4]). *For each regular process P over $A \cup \{\delta\}$ there is a finite extension B of A such that P can be expressed in $\mathrm{ACP}_\tau(B,\gamma)$ with iteration and handshaking only, and the actions in A not subject to communication.*

Proof. Let the regular process P_1 be given by $P_i = \sum_{j=1}^{n}(\alpha_{i,j} \cdot P_j) + \beta_i$ where $\alpha_{i,j}$ and β_i are finite sums of actions or δ.

Define B as the extension of A with the following $3 + 2n$ fresh actions:

$$in,\ r_{stop},\ s_{stop},\ \text{and}\ r_j, s_j \quad (j = 1,\ldots,n).$$

Let $\gamma(r_j,s_j) \stackrel{\text{def}}{=} \gamma(r_{stop},s_{stop}) \stackrel{\text{def}}{=} in$ be the only communications defined (handshaking). As to provide some intuition, these actions model the following behavior:

s_j: instruct the jth process to start,
r_j: read instruction to start the jth process,
s_{stop}: order termination, and
r_{stop}: receive the order to terminate.

Let $H = \{r_{stop},s_{stop}\} \cup \{r_j,s_j \mid i = 1,\ldots,n\}$ and consider the following processes:

$$\sum_{j=1}^{n}(\alpha_{i,j} \cdot s_j) + \beta_i \cdot s_{stop} \quad \text{abbreviated by } G_i \text{ for } i = 1,\ldots,n$$

$$\sum_{j=1}^{n}(r_j \cdot s_j)^*(r_{stop} \cdot s_{stop}) \quad \text{abbreviated by } Mem$$

$$\sum_{j=1}^{n}(r_j \cdot G_j)^* r_{stop} \quad \text{abbreviated by } P.$$

We derive:

$$\begin{aligned}
\partial_H(G_i \cdot P \parallel Mem) &= \partial_H\left(\left(\sum_{j=1}^{n}(\alpha_{i,j} \cdot s_j \cdot P) + \beta_i \cdot s_{stop} \cdot P\right) \parallel Mem\right)\\
&= \sum_{j=1}^{n}(\alpha_{i,j} \cdot \partial_H(s_j \cdot P \parallel Mem)) + \beta_i \cdot \partial_H(s_{stop} \cdot P \parallel Mem)\\
&= \sum_{j=1}^{n}(\alpha_{i,j} \cdot in \cdot \partial_H(P \parallel s_j \cdot Mem)) + \beta_i \cdot in \cdot \partial_H(P \parallel s_{stop})\\
&= \sum_{j=1}^{n}(\alpha_{i,j} \cdot in \cdot in \cdot \partial_H(G_j \cdot P \parallel Mem)) + \beta_i \cdot in \cdot in.
\end{aligned}$$

Consequently, $\tau_{\{in\}} \circ \partial_H(G_i \cdot P \parallel Mem)$ satisfies the equations for P_i $(i = 1,\ldots,n)$. By the principle RSP (a conditional rule, stating that each guarded recursive specification has a unique solution per variable, see e.g. [3]), it follows that $P_i = \tau_{\{in\}} \circ \partial_H(G_i \cdot P \parallel Mem)$ $(i = 1,\ldots,n)$. □

2.2. Generalized merge, expansion and alphabet axioms

The *generalized merge* $[\,\|_{i\in I} P_i]$ 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 is justified by commutativity and associativity of $\|$ (SC). If I is a singleton, say $I = \{i_1\}$,

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

which can be useful in inductive proofs. In some cases it is convenient to use the notation

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

A basic result is the following.

Lemma 2.2 (Merge Lemma).

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

Proof. By induction on n.

$n = 1$. We derive

$$\begin{aligned} x(\tau y \| z) &= x\tau(\tau y \| z) \\ &= x(\tau\tau y \| z) \\ &= x(\tau y \| z) \\ &= x\tau(y \| z) \\ &= x(y \| z). \end{aligned}$$

Notice that by commutativity of $\|$ we obtain $x(\tau y \| \tau z) = x(y \| \tau z) = x(y \| z)$.

$n > 1$. Let $Z = \left(\left[\mathop{\|}_{i=2}^{n} y_i\right] \| z\right)$.

We derive

$$\begin{aligned} x\left(\left[\mathop{\|}_{i=1}^{n} \tau y_i\right] \| z\right) &= x\left(\tau y_1 \| \left(\left[\mathop{\|}_{i=2}^{n} \tau y_i\right] \| z\right)\right) \\ &\overset{(\text{case } n=1)}{=} x\left(\tau y_1 \| \tau\left(\left[\mathop{\|}_{i=2}^{n} \tau y_i\right] \| z\right)\right) \\ &\overset{IH}{=} x(\tau y_1 \| \tau Z) \\ &\overset{(\text{case } n=1)}{=} x(y_1 \| Z). \qquad \square \end{aligned}$$

Furthermore, in the setting of handshaking we can use the *Expansion Theorem* (cf. [3]):

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)	$\alpha(x^* y) = \alpha(x) \cup \alpha(y)$

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

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

$$\text{for } n \geqslant 3: \quad \left[\parallel_{i=1}^{n} P_i\right] = \sum_{j=1}^{n} P_j \lfloor\!\lfloor \left[\parallel_{i \in \{1,\ldots,n\} \setminus \{j\}} P_i\right] + \sum_{j=2}^{n} \sum_{k=1}^{j-1} (P_j \mid P_k) \lfloor\!\lfloor \left[\parallel_{i \in \{1,\ldots,n\} \setminus \{j,k\}} P_i\right].$$

Under certain conditions the scope or action sets I, H of τ_I and ∂_H applications can be changed. These conditions always depend on the *alphabet* of a process: the set of atomic actions it can execute. In Table 2 we give some axioms, where $\alpha(P) \subseteq A$ is the alphabet of process P. Except for AB6, these axioms stem from [2].
Starting from the alphabet of a process, the *conditional alphabet axioms* in Table 3 (taken from [2]) give conditions for changing scope or action sets I, H of τ_I and ∂_H applications. Here $B \mid C$ for $B, C \subseteq A$ denotes the subset $\{a \in A \mid a = \gamma(b, c)$ for some $b \in B, c \in C\}$.

3. Data and process prefixing

In this section we discuss the way in which data and value-passing are specified, and spell out an example on value-passing. Then we introduce an operation and axioms for process prefixing, and apply these to the value-passing example.

3.1. Data and value-passing

In order to reason about processes that manipulate data, we need some minimal assumptions about the data involved: computability in the sense of [9], with only total functions and decidable equality. We adopt a simple specification paradigm for data and actions parameterized with data, which originates from μCRL [15]. Data are used in two ways: in *data-parametric sums* and in *communications*.

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 a be typed as an action that can carry values of type $\mathbb{N}$ (the natural numbers) and of type $\mathbb{N} \times \mathbb{N}$. So $a(0), a(1), \ldots, a(0,0), \ldots$ are considered actions. An example of a data-parametric sum is the expression $\sum(v : \mathbb{N}, a(v))$, denoting a process that for an arbitrary value n of $\mathbb{N}$ can once perform $a(n)$ after which it is terminated. A typical use of this construct is

$$\sum(v : \mathbb{N}, a(v) \cdot a(v, v+v)),$$

which represents the infinite summation $a(0) \cdot a(0,0) + a(1) \cdot a(1,2) + a(2) \cdot a(2,4) + \cdots$. 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 [13, 14]. In particular, these comprise α-conversion and axioms to change its scope.

We further adopt the usual send–read communication paradigm as defined in Table 4. Here the idea is that i is a channel or port identifier, and action $s_i(t)$ models the *sending* of a data value t along port i. An action $r_i(t)$ models the *reading* of the particular value t along channel i. We assume that the communications defined in Table 4 are the only communications defined; in particular this means that the handshaking paradigm is satisfied.

With help of send–read communication and the encapsulation operations ∂_H one can easily model value-passing (cf. [20]). For a small, typical example consider

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

a process that is willing to receive any natural along channel 1, and $s_1(5) \cdot S$, 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

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

where the second identity follows from the axioms of $\mathrm{ACP}^{\tau}(A, \gamma)$ and those for the $\sum$-operation. So, by encapsulation, the action $s_1(5)$ enforces the communication action $c_1(5)$. Hence, this communication action models the *value-passing* of 5 along channel 1

between the two parallel components of $\partial_{\{r_1,s_1\}}(R \parallel s_1(5)\cdot S)$. 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 [12, p. 69].

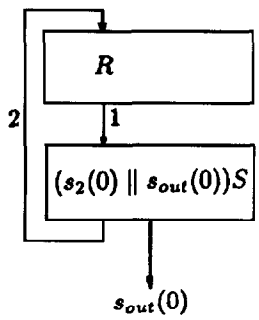
We finish this section with an example that describes a simple, one-module network.

Example 3.1. Consider the following two processes, abbreviated by R and S:

$$R = \sum(v:\mathbb{N},\ r_2(v)\cdot s_1(v+1))^*\delta,$$
$$S = \sum(v:\mathbb{N},\ r_1(v)(s_2(v) \parallel s_{out}(v)))^*\delta.$$

Let $H = \{r_1, s_1, r_2, s_2\}$. The behavior of $\partial_H(R \parallel (s_2(0) \parallel s_{out}(0))\cdot S)$ can be analyzed and visualized as follows (note that actions $s_{out}(j)$ cannot be involved in a communication):

$$\begin{aligned}
&\partial_H(R \parallel (s_2(0) \parallel s_{out}(0))\cdot S)\\
&\quad = c_2(0)\cdot \partial_H(s_1(1)\cdot R \parallel s_{out}(0)\cdot S)\\
&\qquad + s_{out}(0)\cdot \partial_H(R \parallel s_2(0)\cdot S)\\
&\quad = c_2(0)\cdot s_{out}(0)\cdot \partial_H(s_1(1)\cdot R \parallel S)\\
&\qquad + s_{out}(0)\cdot c_2(0)\cdot \partial_H(s_1(1)\cdot R \parallel S)\\
&\quad = c_2(0)\cdot s_{out}(0)\cdot c_1(1)\cdot \partial_H(R \parallel (s_2(1) \parallel s_{out}(1))\cdot S)\\
&\qquad + s_{out}(0)\cdot c_2(0)\cdot c_1(1)\cdot \partial_H(R \parallel (s_2(1) \parallel s_{out}(1))\cdot S).
\end{aligned}$$



Let $I = \{c_1, c_2\}$ and let $P(n) = \tau_I \circ \partial_H(R \parallel (s_2(n) \parallel s_{out}(n))\cdot S)$ for some $n \in \mathbb{N}$. From the derivation above it follows that the one-module network $P(n)$ satisfies

$$P(n) = \tau\cdot s_{out}(n)\cdot P(n+1) + s_{out}(n)\cdot P(n+1).$$

Hence $\tau\cdot P(n) = \tau\cdot s_{out}(n)\cdot P(n+1)$, expressing that $\tau\cdot P(n)$ outputs the infinite stream

$$\tau\cdot s_{out}(n)\cdot s_{out}(n+1)\cdot s_{out}(n+2)\cdot\cdots.$$

3.2. Process prefixing

Let D be some data type. We consider the process prefix operation, notation ;, and early-read actions $er_i(v)$ with i a channel or port identifier and v a variable of type D (cf. [1]). The early-read axiom scheme, parameterized with data type D, is

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

(so v may occur in an instantiation of x).[3] As an example consider

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

[3] 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 [20].

Table 5
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)	$(x \cdot y); z = x; (y; z)$

which is an expression without free data-variables. Furthermore,

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

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 $r_i : D_1 \times \cdots \times D_n$ 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. In the μCRL setting, this implies that an early read over *pairs* of values corresponds with two $\sum$-applications (which commute [13, 14]), e.g., for $r_k : D \times \mathbb{N}$, function $F : D \times \mathbb{N} \to \mathbb{N}$, and action $s_l : Nat$ we obtain

$$er_k(v, w); s_l(F(v, w)) = \sum(v : D, \sum(w : \mathbb{N}, r_k(v, w) \cdot s_l(F(v, w)))).$$

Alternatively, one can consider a setting with variables over products of the data types involved. Note that for the *er* actions we use *globally* typed variables.

Let $\mathrm{ACP}^{\tau}_{er}(A, \gamma)$ be the extension of $\mathrm{ACP}^{\tau}(A, \gamma)$ with early-read actions and process prefixes as introduced above. A particular – and intended – consequence of 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):

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

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,

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

$$= \delta + er_2(w); \partial_{\{r_1\}}(P_2)$$
$$= er_2(w); \partial_{\{r_1\}}(P_2).$$

Example 3.2. The process $P(n)$ from Example 3.1 can now be specified by

$$\tau_I \circ \partial_H(((er_2(v); s_1(v+1))^*\delta) \| (s_2(n) \| s_{out}(n)) \cdot (er_1(v); (s_2(v) \parallel s_{out}(v))^*\delta))$$

with v a variable of type $\mathbb{N}$.

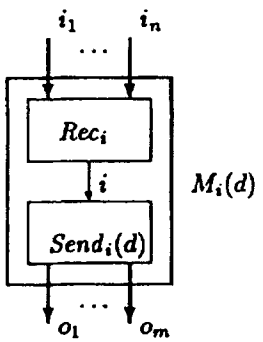
4. Modules and networks, specification

In this section we propose a specification format for *modules*, elementary processors of data. Next we introduce *networks* as a format for the parallel execution of such modules. In fact, the process $P(n)$ defined in Example 3.1 exemplifies the most simple type of network that we consider, containing one module. Our modeling is based on [22], in which SCAs (synchronous concurrent algorithms) are analyzed.

4.1. Modules

A module M_i is supposed to contain a value, a (positive) number n of input channels, and a (positive) number m of output channels. To keep things simple, we first restrict ourselves to a setting with only one data type D. The computational functionality of a module M_i is characterized by a (total) value function $F_i : D^n \to D$. We specify a module $M_i(d)$ with current value d, input channels $i_1, \ldots, i_n$ and output channels $o_1, \ldots, o_m$ by means of two iterative processes. The first one of these defines the *receive*-part Rec_i of the module (modeling the read actions), the second its *send*-part $Send_i(d)$ (ready to send the value d along the ports $o_1, \ldots, o_m$). These two parts communicate along some channel i, internal to module M_i. The computational functionality of module M_i is modeled in the (internal) s_i-action of Rec_i, which can take place after all parallel read actions of Rec_i have been executed. This yields the following specification and picture of $M_i(d)$:

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



Now assume that $\{i_1,\ldots,i_n\}\cap\{o_1,\ldots,o_m\}=\emptyset$. It then follows that $M_i(d)$ has a process prefix

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

(this is a consequence of Theorem 6.1). After having read certain values $d_1,d_2,\ldots,d_n$ along channels $i_1,\ldots,i_n$, and having sent d along ports $o_1,\ldots,o_m$, the module's current value is updated to $F_i(d_1,\ldots,d_n)$ (by a communication along channel i, renamed into the silent action τ), and the next process prefix is ready to be performed:

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

The case that a module reads its own value as an input, i.e. $\{i_1,\ldots,i_n\}\cap\{o_1,\ldots,o_m\}\neq\emptyset$, is called *feedback*. Per module, at most one feedback channel is allowed in our setting.

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

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

Henceforth, $M_i(d)=Rec_i\parallel_i Send_i(d)$.

It is evident that the specific typing of the channels (i.e., of the read and send actions) is not relevant, as long as the function F_i is compatible with it. Therefore, we further consider a many-sorted setting. We assume that each variable is uniquely typed.

4.2. Networks

A network is just a collection of modules, in which the read/send connections respect the typing of the corresponding modules. A general restriction is that there is *at most one* channel for transmission of data from a module to a module (which may be the sending module). In this paper we consider networks of the form

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

where the ∂_H applications model value-passing synchronizations between the modules $M_1,\ldots,M_n$. We further distinguish the following network characteristics:

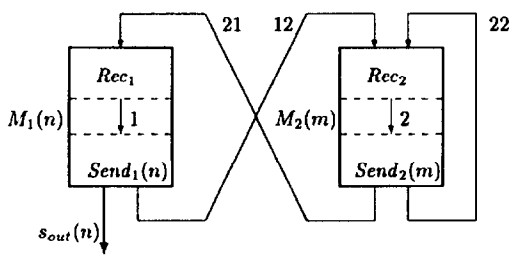
Definition 4.1. (1) The *Input/Output* of a network, *I/O* for short, denotes the network's *external* actions, i.e., read or send actions that have no communication partner within the network.

(2) A network is an *I/O network* if it has a positive number of external actions. It is *single-output* if its *I/O* consists of exactly one output action, which will be referred to as $s_{out}(\cdots)$.

(3) A network determines its underlying graph by taking its module identifiers as nodes, and its internal communication channels as (undirected) edges. A network is *connected* if its underlying graph is connected.[4]

Below we give an example for computing a Fibonacci sequence using a connected single-output network consisting of modules M_1 and M_2. This example also illustrates the reason for computing the next 'current value' in a module only after all the current input *and output* actions have been performed: the latter may have to communicate with some of the module's input actions (in the case of feedback).

Example 4.2 (*Fibonacci Network*). Recall the Fibonacci sequence defined by $f_0 = f_1 = 1$, $f_{n+2} = f_n + f_{n+1}$. Consider the following network in which all data to be transmitted are of type $\mathbb{N}$. 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 $M_2(m)$:

$$M_1(n) = Rec_1 \parallel_1 Send_1(n), \qquad M_2(m) = Rec_2 \parallel_2 Send_2(m),$$
$$Rec_1 = (er_{21}(v); s_1(v))^* \delta, \qquad Rec_2 = ((er_{12}(v_1) \parallel er_{22}(v_2)); s_2(v_1 + v_2))^* \delta,$$
$$Send_1(n) = (s_{out}(n) \parallel s_{12}(n)) \cdot Send_1, \qquad Send_2(m) = (s_{21}(m) \parallel s_{22}(m)) \cdot Send_2,$$
$$Send_1 = (er_1(v); (s_{out}(v) \parallel s_{12}(v)))^* \delta, \qquad Send_2 = (er_2(v); (s_{21}(v) \parallel s_{22}(v)))^* \delta.$$

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:

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

[4] Recall: two nodes in a finite, undirected graph are *connected* if there is a path that connects them; the graph is *connected* if each pair of different nodes is connected.

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 5.5 discussed in the following section.

5. Verification of connected, single-output networks

This section leads to a correctness result on connected, single-output networks, quoted here.

Theorem 5.5. *Let $n \geqslant 1$, $\vec{d} = d_1, \ldots, d_n$ be a collection of typed values, and let*

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

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. In 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 theorem can be easily proved.

The reader not interested in the technical details of the proof of Theorem 5.5 can skip the rest of this section (Section 5), in which we propose some uniform notation, and establish various intermediate results that we use for the proof of the theorem quoted above.

5.1. Notational conventions

First we fix some notation. We consider data types $D, D_1, D_2, \ldots$ and functions $F_1, F_2, \ldots$ over these. For an I/O network

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

of size n with $d_i \in D_i$, we assume that module $M_j(d_j) = Rec_j \parallel_j Send(d_j)$, so the (internal) channel between the receive and send part of module M_j has identifier j.

We further assume that Rec_j has input channels indexed from a (non-empty) set R_j and value function F_j, and that $Send_j$ has output channels indexed from a (non-empty) set S_j. Observe that both these sets are disjoint with $\{1,\ldots,n\}$, the set of internal channels of the modules $M_1,\ldots,M_n$, respectively.

As to characterize typical states in the execution of a network, we introduce some abbreviations. The receive-part Rec_l of a module M_l either has received all data of the appropriate type, or has not ($\vec{x}$ below is typed as the domain of F_l):

$$Rec_l(R',\vec{x}) \stackrel{\text{def}}{=} \begin{cases} s_l(F_l(\vec{x})) \cdot Rec_l & \text{if } R' = \emptyset \text{ and } x_i \in D_{j_i} \\ \left(\left[\parallel_{m \in R'} er_m(v_m) \right] ; s_l(F_l(\vec{x})) \right) \cdot Rec_l & \begin{cases} \text{if } R' \subseteq R_l, R \neq \emptyset, \\ m \in R' \Rightarrow x_m \equiv v_m, \\ \text{and } m \notin R' \Rightarrow x_m \in D_{j_m}. \end{cases} \end{cases}$$

For the send-part $Send_k$ of a module M_k we introduce

$$Send_k(S',d) \stackrel{\text{def}}{=} \begin{cases} Send_k & \text{if } S' = \emptyset \\ \left[\parallel_{m \in S'} s_m(d) \right] \cdot Send_k & \text{if } S' \subseteq S_k \text{ and } S' \neq \emptyset. \end{cases}$$

Observe that by our definition of modules (see Section 4.1) and the abbreviations introduced above, $Send_k(d_k) \equiv Send_k(S_k, d_k)$.

5.2. Some network properties

In this section we establish four intermediate results, which we use in the proof of our correctness result. The first of these states that a value-passing communication in a network is not observable in a $\tau \cdot [\ ]$ context.

Lemma 5.1. *Let module M_k transmit values to M_l along channel j, and $S' \subseteq S_k$, $R' \subseteq R_l$ and $(R_l \cup S_k) \ni j \notin (R' \cup S')$. Then*

$$\tau \cdot (Rec_l(R' \cup \{j\},\vec{x}) \parallel_j Send_k(S' \cup \{j\},d)) = \tau \cdot (Rec_l(R',\vec{y}) \parallel_j Send_k(S',d)),$$

where $i \neq j \Rightarrow x_i = y_i$ and $y_j = d$. (Note that $S' \cap R' = \emptyset$ and that the case $k = l$, i.e., feedback, is not excluded.)

Proof. By induction on $|R'| + |S'| = N$.

In case $R' = S' = \emptyset$ we are done: the communication along channel j is the only possible action.

In case $N > 0$ and $S' = \emptyset$, $R' \neq \emptyset$, we derive

$$\begin{aligned}
&\tau \cdot (Rec_l(R' \cup \{j\}, \vec{x}) \parallel_j Send_k(\{j\}, d)) \\
&\overset{(*)}{=} \tau \cdot \left(\begin{array}{l} \tau \cdot (Rec_l(R', \vec{y}) \parallel_j Send_k(\emptyset, d)) \\ + \\ \sum_{m \in R'} er_m(v_m); (Rec_l((R' \backslash \{m\}) \cup \{j\}, \vec{x}) \parallel_j Send_k(\{j\}, d)) \end{array} \right) \\
&\overset{IH_1(*)}{=} \tau \cdot \left(\begin{array}{l} \tau \cdot \left(\begin{array}{l} (Rec_l(R', \vec{y}) \parallel_j Send_k(\emptyset, d)) \\ + \\ \sum_{m \in R'} er_m(v_m); (Rec_l(R' \backslash \{m\}, \vec{y}) \parallel_j Send_k(\emptyset, d)) \end{array} \right) \\ + \\ \sum_{m \in R'} er_m(v_m); (Rec_l(R' \backslash \{m\}, \vec{y}) \parallel_j Send_k(\emptyset, d)) \end{array} \right) \\
&\overset{B2}{=} \tau \cdot \left(\begin{array}{l} (Rec_l(R', \vec{y}) \parallel_j Send_k(\emptyset, d)) \\ + \\ \sum_{m \in R'} er_m(v_m); (Rec_l(R' \backslash \{m\}, \vec{y}) \parallel_j Send_k(\emptyset, d)) \end{array} \right) \\
&\overset{(*)}{=} \tau \cdot (Rec_l(R', \vec{y}) \parallel_j Send_k(\emptyset, d)).
\end{aligned}$$

As for $(*)$, first observe that the process prefix of $Rec_l(R' \cup \{j\}, \vec{x})$, i.e.,

$$\left[\parallel_{m \in R' \cup \{j\}} er_m(v_m) \right]$$

is by the Expansion Theorem equal to

$$\sum_{k \in R' \cup \{j\}} er_k(v_k) \cdot \left[\parallel_{m \in (R' \cup \{j\}) \backslash \{k\}} er_m(v_m) \right]$$

so by axiom (PP6) we get $Rec_l(R' \cup \{j\}, \vec{x}) = \sum_{k \in R' \cup \{j\}} er_k(v_k)$; $Rec_l((R' \cup \{j\}) \backslash \{k\}, \vec{x})$. Furthermore, it is essential that a $Send_k(S, d)$ process does not contain free variable v_m. For this reason we can use the identity

$$er_m(v_m); X \mathbin{\lfloor\!\lfloor} Send_k(S, d)) = er_m(v_m); (X \parallel Send_k(S, d)),$$

which is a particular instance of a μCRL axiom[5] on getting $\ldots \mathbin{\lfloor\!\lfloor} Q$ into the scope of a $\sum$-application:

$$\begin{aligned}
\sum (v_m : D,\ r_m(v_m) \cdot P) \mathbin{\lfloor\!\lfloor} Q &= \sum (v_m : D,\ r_m(v_m) \cdot P \mathbin{\lfloor\!\lfloor} Q) \\
&\left(= \sum (v_m : D,\ r_m(v_m) \cdot (P \parallel Q)) = er_m(v_m); (P \parallel Q) \right)
\end{aligned}$$

where Q may not contain v_m as a free variable.

[5] The axiom SUM6, see [13, 14].

The case $N > 0$ and $R' = \emptyset$, $S' \neq \emptyset$ is similar. Let $R' \neq \emptyset \neq S'$. We derive

$$\tau \cdot (Rec_l(R' \cup \{j\}, \vec{x}) \parallel_j Send_k(S' \cup \{j\}, d))$$

$$\overset{(*)}{=} \tau \cdot \left(\begin{array}{l} \tau \cdot (Rec_l(R', \vec{y}) \parallel_j Send_k(S', d)) \\ + \\ \sum_{m \in R'} er_m(v_m); (Rec_l((R' \backslash \{m\}) \cup \{j\}, \vec{x}) \parallel_j Send_k(\{j\}, d)) \\ + \\ \sum_{m \in S'} s_m(d) \cdot (Rec_l(R' \cup \{j\}, \vec{x}) \parallel_j Send_k((S' \backslash \{m\}) \cup \{j\}, d)) \end{array} \right)$$

$$\overset{IH,(*)}{=} \tau \cdot \left(\begin{array}{l} \tau \cdot \left(\begin{array}{l} (Rec_l(R', \vec{y}) \parallel_j Send_k(S', d)) \\ + \\ \sum_{m \in R'} er_m(v_m); (Rec_l(R' \backslash \{m\}, \vec{y}) \parallel_j Send_k(S', d)) \\ + \\ \sum_{m \in S'} s_m(d) \cdot (Rec_l(R', \vec{y}) \parallel_j Send_k((S' \backslash \{m\}) \cup \{j\}, d)) \end{array} \right) \\ + \\ \sum_{m \in R'} er_m(v_m); (Rec_l(R' \backslash \{m\}, \vec{y}) \parallel_j Send_k(S', d)) \\ + \\ \sum_{m \in S'} s_m(d) \cdot (Rec_l(R', \vec{y}) \parallel_j Send_k((S' \backslash \{m\}) \cup \{j\}, d)) \end{array} \right)$$

$$\overset{B2,(*)}{=} \tau \cdot (Rec_l(R', \vec{y}) \parallel_j Send_k(S', d)). \quad \square$$

As a corollary we obtain that the particular order of the possible value-passing communications in a network is not relevant.

Corollary 5.2. *For an I/O network* $N(\vec{d}) = \tau_I \circ \partial_H \left(\left[\|_{i=1}^n M_i(d_i) \right] \right)$ *it holds that*

$$\tau \cdot N(\vec{d}) = \tau \cdot \tau_I \circ \partial_H \left(\left[\|_{i=1}^n (Rec_i(R_i^{out}, \vec{x}_i) \parallel_i Send_i(S_i^{out}, d_i)) \right] \right)$$

where R_i^{out} *represents the input channels of module* M_i *that are not connected in the network, and* S_i^{out} *the non-connected output channels.*

Proof. First observe that by the alphabet axioms, one can interchange synchronization of the internal communication of a module M_j (along channel j and enforced by $\parallel_j$, modeling its value-update) and a value-passing communication as considered in the previous lemma. Now apply Lemma 5.1 on all internal value-passing channels of N (including feedback channels), while using the Merge Lemma 2.2 when appropriate. It is clear that the order in which this is done is not relevant: each two possible value-passing communications commute. $\square$

Also we shall need the following result, stating that the value-update communications of the modules of a network are not observable in a $\tau \cdot [\,]$ context.

Lemma 5.3. *For an I/O network* $N(\vec{d}) = \tau_I \circ \partial_H([\,\|_{i=1}^n M_i(d_i)])$ *it holds that*

$$\tau \cdot \tau_I \circ \partial_H\left(\left[\,\|_{i=1}^n (Rec_i(\emptyset, \vec{d}_i) \,\|_i\, Send_i)\right]\right) = \tau \cdot N(F_1(\vec{d}_1), \ldots, F_n(\vec{d}_n)).$$

where $\vec{d}_i$ *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.*

Proof. Immediately from the Merge Lemma 2.2. □

We further restrict attention to networks that are *connected.* The following intermediate result (the last one we need) states that a connected I/O network can initially perform at most a finite number of internal actions, i.e., τ-steps, arriving in a unique state in which no further internal steps are possible, and an I/O action must be performed. This depends on connectedness: consider the network discussed in Example 3.2. Extending this one with a single, isolated module of the form

(only performing feedback) yields a network in which the internal feedback-activity sketched above – resulting in τ-steps – can be performed in each state.

Lemma 5.4 (External Action Lemma). Let $N(\vec{d}) = \tau_I \circ \partial_H([\,\|_{i=1}^n M_i(d_i)])$ be a connected I/O network. Then there is an expression $\widetilde{N}(\vec{d})$ such that
1. $\tau \cdot N(\vec{d}) = \tau \cdot \widetilde{N}(\vec{d})$, and
2. $\widetilde{N}(\vec{d})$ cannot perform an internal action.

Proof. In case the size of $N(\vec{d})$ is 1, the statement is trivial: at most one internal 'feedback action' can be performed. Assume $N(\vec{d})$ contains at least two modules. By Corollary 5.2 we have

$$\tau \cdot N(\vec{d}) = \tau \cdot \tau_I \circ \partial_H\left(\left[\,\|_{i=1}^n (Rec_i(R_i^{out}, \vec{x}_i) \,\|_i\, Send_i(S_i^{out}, d_i))\right]\right).$$

Because $N(\vec{d})$ performs I/O, at least one of $R_i^{out} \cup S_i^{out}$ is non-empty and the corresponding module can only perform an external action. In case all are non-empty, we have found our $\widetilde{N}(\vec{d})$. If not, partition $\{1, \ldots, n\}$ into *Extern* and *Intern* such that

$$j \in Intern \;\Leftrightarrow\; R_j^{out} \cup S_j^{out} = \emptyset.$$

Notice that $\emptyset \subsetneqq Intern \subsetneqq \{1,\ldots,n\}$ in this case. We derive

$$\tau \cdot N(\vec{d}) = \tau \cdot \tau_I \circ \delta_H \left(\begin{bmatrix} \|_{i \in Intern} (Rec_i(\emptyset, \vec{d}_i) \|_i Send_i(\emptyset, d_i)) \\ \| \\ \|_{i \in Extern} (Rec_i(R_i^{out}, \vec{x}_i) \|_i Send_i(S_i^{out}, d_i)) \end{bmatrix} \right)$$

$$\overset{(5.3)}{=} \tau \cdot \tau_I \circ \delta_H \left(\begin{bmatrix} \|_{i \in Intern} (Rec_i(R_i, \vec{v}_i) \|_i Send_i(S_i, F_i(\vec{d}_i))) \\ \| \\ \|_{i \in Extern} (Rec_i(R_i^{out}) \vec{x}_i \|_i Send_i(S_i^{out}, d_i)) \end{bmatrix} \right).$$

Possible internal steps are between the *Intern*-modules. We repeat a similar procedure on the set *Intern*. Exhaust all internal communications between the *Intern*-modules according to Lemma 5.1. Let the resulting index sets be $R_i^{out,1}$ and $S_i^{out,1}$. By connectedness, there is at least one connection with an *Extern*-module, so the corresponding $R_i^{out,1} \cup S_i^{out,1}$ is non-empty. In case all *Intern*-modules are connected with an *Extern*-module we have found our $\widetilde{N}(\vec{d})$. If not, partition *Intern* into *Intern*1 and *Intern*2 such that

$$j \in Intern1 \Leftrightarrow R_i^{out,1} \cup S_i^{out,1} = \emptyset.$$

By assumption and connectedness, $\emptyset \subsetneqq Intern1 \subsetneqq Intern$. Let $\vec{e} = e_1,\ldots,e_n$ where $e_i = F_i(\vec{d}_i)$. Consider the derivation in Table 6. Possible internal steps are between the *Intern*1-modules. Now we can repeat a similar procedure on the set *Intern*1. Continuing in this fashion, we end up with an expression that initially only allows external actions: each further partition yields a smaller set of possible internal actions. □

5.3. Correctness of connected, single-output networks

Using the results from the preceding section, we are able to give a short proof of the correctness result quoted before.

Theorem 5.5. *Let $n \geqslant 1$, $\vec{d} = d_1,\ldots,d_n$ be a collection of typed values, and let*

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

be a network that is single-output and connected, 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.

Table 6
Proving the External Action Lemma 5.4.

$$\tau\cdot N(\vec{d}) = \tau\cdot\tau_I\circ\delta_H\left(\begin{array}{l}\left[\begin{array}{l}\|\\ {\scriptstyle i\in Intern}\end{array}(Rec_i(R_i,\vec{v}_i)\parallel_i Send_i(S_i,F_i(\vec{d}_i)))\right]\\ \|\\ \left[\begin{array}{l}\|\\ {\scriptstyle i\in Extern}\end{array}(Rec_i(R_i^{out})\vec{x}_i\parallel_i Send_i(S_i^{out},d_i))\right]\end{array}\right)$$

$$=\tau\cdot\tau_I\circ\delta_H\left(\begin{array}{l}\left[\begin{array}{l}\|\\ {\scriptstyle i\in Intern1}\end{array}(Rec_i(\emptyset,\vec{e}_i)\parallel_i Send_i(\emptyset,e_i))\right]\\ \|\\ \left[\begin{array}{l}\|\\ {\scriptstyle i\in Intern2}\end{array}(Rec_i(R_i^{out,1},\vec{y}_i)\parallel_i Send_i(S_i^{out,1},e_i))\right]\\ \|\\ \left[\begin{array}{l}\|\\ {\scriptstyle i\in Extern}\end{array}(Rec_i(R_i^{out},\vec{x}_i)\parallel_i Send_i(S_i^{out},d_i))\right]\end{array}\right)$$

$$\overset{(5.3)}{=}\tau\cdot\tau_I\circ\delta_H\left(\begin{array}{l}\left[\begin{array}{l}\|\\ {\scriptstyle i\in Intern1}\end{array}(Rec_i(R_i,\vec{v}_i)\parallel_i Send_i(S_i,F_i(\vec{e}_i)))\right]\\ \|\\ \left[\begin{array}{l}\|\\ {\scriptstyle i\in Intern2}\end{array}(Rec_i(R_i^{out,1},\vec{y}_i)\parallel_i Send_i(S_i^{out,1},e_i))\right]\\ \|\\ \left[\begin{array}{l}\|\\ {\scriptstyle i\in Extern}\end{array}(Rec_i(R_i^{out},\vec{x}_i)\parallel_i Send_i(S_i^{out},d_i))\right]\end{array}\right).$$

Proof. We distinguish the cases $n=1$ and $n>1$.

$n=1$. In this case $M_1(d_1)$ necessarily is a module with one feedback channel and no other inputs, say

$$M_1(d_1)=((er_2(v);s_1(F_1(v)))^*\delta\parallel_1(s_2(d_1)\|s_{out}(d_1))\cdot((er_1(v);(s_2(v)\parallel s_{out}(v)))^*\delta))$$

with feedback channel 2. A typical case that proves the theorem is spelled out in Examples 3.1 and 3.2 (apart from some trivial applications of the alphabet axioms).

$n>1$. Partition the underlying graph of $\{M_2,\ldots,M_n\}$ (see Definition 4.1.3) into connected subgraphs of maximal size. Notice that this partition is *unique*. Let $C1,\ldots,Ck$ represent this partition ($Ci\subseteq\{2,\ldots,n\}$ and $1\leqslant k\leqslant n-1$). Consider for $j\in\{1,\ldots,k\}$ the network

$$\tau_{Ij}\circ\partial_{Hj}\left(\left[\begin{array}{l}\|\\ {\scriptstyle i\in Cj}\end{array}M_i(d_i)\right]\right),$$

where Ij,Hj refer to all channels between the Cj-indexed modules. Observe that each such network is connected and performs I/O with M_1 only.

Let $\widetilde{d_j}$ be the sequence $(d_i)_{i\in Cj}$ and

$$N_j(\widetilde{d_j}) = \tau_{Ij} \circ \partial_{Hj}\left(\left[\Vert_{i\in Cj} M_i(d_i)\right]\right).$$

Furthermore, let $\vec{e} = e_1,\ldots,e_n$ where $e_i = F_i(\vec{d}_i)$. We derive (see explanation below):

$$\tau\cdot N(\vec{d}) \overset{(A)}{=} \tau\cdot\tau_I\circ\partial_H\left(\left[\Vert_{i=1}^{n}(Rec_i(R_i^{out},\vec{x}_i)\Vert_i Send_i(S_i^{out},d_i))\right]\right)$$

$$\overset{(B)}{=} \tau\cdot\tau_I\circ\partial_H\left(\begin{array}{l}(Rec_1(\emptyset,\vec{d}_1)\Vert_1 Send_1(\{out\},d_1))\\ \Vert\\ \left[\Vert_{j=1}^{k}\tau_{Ij}\circ\partial_{Hj}\left(\left[\Vert_{i\in Cj}(Rec_i(\emptyset,\vec{d}_i)\Vert_i Send_i)\right]\right)\right]\end{array}\right)$$

$$\overset{(C)}{=} \tau\cdot\tau_I\circ\partial_H\left(\begin{array}{l}(s_1(F_1(\vec{d}_1))\cdot Rec_1\Vert_1 s_{out}(d_1)\cdot Send_1)\\ \Vert\\ \left[\Vert_{j=1}^{k}\tau\cdot N_j(\widetilde{e_j})\right]\end{array}\right)$$

$$\overset{(D)}{=} \tau\cdot\tau_I\circ\partial_H\left(\begin{array}{l}(s_1(F_1(\vec{d}_1))\cdot Rec_1\Vert_1 s_{out}(d_1)\cdot Send_1)\\ \Vert\\ \left[\Vert_{j=1}^{k}\tau\cdot\widetilde{N_j}(\widetilde{e_j})\right]\end{array}\right)$$

$$\overset{(E)}{=} \tau\cdot s_{out}(d_1)\cdot\tau_I\circ\partial_H\left(\begin{array}{l}(s_1(F_1(\vec{d}_1))\cdot Rec_1\Vert_1 Send_1)\\ \Vert\\ \left[\Vert_{j=1}^{k}\tau\cdot\widetilde{N_j}(\widetilde{e_j})\right]\end{array}\right)$$

$$\overset{(F)}{=} \tau\cdot s_{out}(d_1)\cdot\tau_I\circ\partial_H\left(\begin{array}{l}M_1(F_1(\vec{d}_1))\\ \Vert\\ \left[\Vert_{j=1}^{k}\tau\cdot N_j(\widetilde{e_j})\right]\end{array}\right)$$

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

Explanation: in most steps we tacitly use the Merge Lemma 2.2. Furthermore,

(A) Here we apply Corollary 5.2. Notice that all R_i^{out} and S_i^{out} sets are empty, except for S_1^{out} consisting of *out*.

(B) Here we apply the alphabet axioms.

(C) Here we replace $Rec_1(\emptyset,\vec{d}_1)$ and $Send_1(\{out\},d_1)$ by their definitions, and apply Lemma 5.3.

(D) Here we apply Lemma 5.4 on all $N_j(\widetilde{e_j})$.
(E) Because each action of each $\widetilde{N_j(\widetilde{e_j})}$ is external with respect to Ij, Hj and can only perform an (internal) communication with M_1, the only possible step is the $s_{out}(d_1)$-action.
(F) The internal step of M_1 and Lemma 5.4 are applied.
(G) The alphabet axioms are applied. □

6. Generalizations, specifications and verifications

We can relax the conditions under which the execution of a network satisfies a single process prefix, followed by a recursive update of its data state. Output may be modified or multiplied, and a restricted form of external input can be allowed. In the rest of this section we make this precise.

6.1. Output modification

Our first generalizations concern 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 of the form $s_{out}(d)$. We call this *output modification* of the *out*-channel.

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

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

has more than one output channel, and that I is such that all extra output channels are hidden. Then Theorem 5.5 still is applicable. We prove this below. Notice that it is sufficient to show that one extra, *hidden* output action does not change the external behavior of the network when considered in a $\tau \cdot [\,]$ context. Now assume *extra* is the identifier of such an additional, hidden output channel that originates from module M_k. So $s_{extra} \in I$. Let

$$Send_k = \left(er_k(v); \left[\|_{i \in S_k} s_i(v) \right] \right)^* \delta,$$

$$Send_k^{extra} = \left(er_k(v); \left[\|_{i \in S_k \cup \{extra\}} s_i(v) \right] \right)^* \delta.$$

By induction on $|S_k|$ (recall: $|S_k| > 0$) it follows easily that

$$\tau \cdot \tau_{\{extra\}}\left(\left[\mathop{\|}_{i\in S_k\cup\{extra\}} s_i(t)\right]\right) = \tau \cdot \left[\mathop{\|}_{i\in S_k} s_i(t)\right].$$

Furthermore, $\tau_{\{extra\}}(Send_k^{extra}) = Send_k$ because the $\tau_{\{extra\}}$ application distributes over all operations involved:

$$\begin{aligned}
\tau_{\{extra\}}(Send_k^{extra}) &= \tau_{\{extra\}}\left(\sum\left(v:D,\ r_k(v)\cdot\left[\mathop{\|}_{i\in S_k\cup\{extra\}} s_i(v)\right]\right)*\delta\right)\\
&= \tau_{\{extra\}}\left(\sum\left(v:D,\ r_k(v)\cdot\left[\mathop{\|}_{i\in S_k\cup\{extra\}} s_i(v)\right]\right)\right)*\tau_{\{extra\}}(\delta)\\
&= \sum\left(v:D,\ \tau_{\{extra\}}\left(r_k(v)\cdot\left[\mathop{\|}_{i\in S_k\cup\{extra\}} s_i(v)\right]\right)\right)*\delta\\
&= \sum\left(v:D,\ r_k(v)\cdot\tau_{\{extra\}}\left(\left[\mathop{\|}_{i\in S_k\cup\{extra\}} s_i(v)\right]\right)\right)*\delta\\
&= \sum\left(v:D,\ r_k(v)\cdot\left[\mathop{\|}_{i\in S_k} s_i(v)\right]\right)*\delta.
\end{aligned}$$

Hence, $\tau \cdot \tau_{\{extra\}}(Send_k^{extra}(d_k)) = \tau \cdot Send_k(d_k)$.

Now let

$$M_k^{extra}(d_k) = (Rec_k \|_k Send_k^{extra}(d_k)).$$

With some applications of the alphabet axioms it follows that

$$\tau \cdot \tau_{\{extra\}}(M_k^{extra}(d_k)) = \tau \cdot M_k(d_k).$$

Finally, let $N^{extra}(\vec{d})$ be obtained from $N(\vec{d})$ by replacing $M_k(d_k)$ with $M_k^{extra}(d_k)$. From the alphabet axioms and the Merge Lemma 2.2 it easily follows that

$$\tau \cdot N^{extra}(\vec{d}) = \tau \cdot N(\vec{d}).$$

Consequently, a correctness characterization for $\tau \cdot N^{extra}(\vec{d})$ can be obtained from Theorem 5.5:

$$\begin{aligned}
\tau \cdot N^{extra}(\vec{d}) &= \tau \cdot N(\vec{d})\\
&\overset{5.5}{=} \tau \cdot s_{out}(d_1)\cdot N(F_1(\vec{d_1}),\ldots,F_n(\vec{d_n}))\\
&= \tau \cdot s_{out}(d_1)\cdot N^{extra}(F_1(\vec{d_1}),\ldots,F_n(\vec{d_n})).
\end{aligned}$$

We further consider a single-output, connected network as one that may contain extra, hidden output channels, and that may return 'modified' output.

6.2. Single-I/O networks

A network 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*. In this section we establish a characterization result for single-I/O networks. This 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 a single output-network to a single-I/O network while preserving a simple correctness characterization.

Including the generalizations from the previous section, we extend our characterization result to the class of connected, single-I/O networks.

Theorem 6.1. *Let $n \geqslant 1$, $\vec{d} = d_1, \ldots, d_n$ be a collection of typed values, and let*

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

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

$$\tau \cdot N(\vec{d}) = \tau \cdot \left[\|_{i \in Extern} a_i(x_i) \right] ; \tau_I \circ \partial_H \left(\begin{array}{l} (s_1(F_1(\vec{x}_1)) \cdot Rec_1 \,\|_1\, Send_1) \\ \| \\ \left[\|_{i=2}^{n} M_i(F_i(\vec{d}_i)) \right] \end{array} \right),$$

where

- *in case $n = 1$, the postfix expression reads $\tau_I \circ \partial_H(s_1(F_1(\vec{x}_1)) \cdot Rec_1 \,\|_1\, Send_1)$.*
- *F_i is the value function of module M_i,*
- *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 $a_i(x_i) \equiv er_i(v_i)$,*
- *for $i > 1$, $\vec{d}_i$ 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,*
- *$\vec{x}_1$ is defined similar, except for its Extern-coordinates (see the third clause).*

Proof. Almost exactly as in the proof of Theorem 5.5. Differences are:

1. In step (B) the expression $Rec_1(\emptyset, \vec{d}_1) \,\|_1\, Send_1(\{out\}, d_1)$ has to be replaced by

   $$Rec_1(In, \vec{d}_1) \,\|_1\, Send_1(Out, d_1)$$

   where $In \cup Out = Extern$, and In (Out) is the index set of the network's external input (output) actions, and
2. In step (E), the complete prefix $[\|_{i \in Extern} a_i(x_i)]$ has to pass the scope of the $\tau_I \circ \partial_H$-application. An application of the Expansion Theorem is helpful here. □

Observe that in case there are no external input actions, we find a straightforward generalization of Theorem 5.5: in a $\tau \cdot [\ ]$ context the network recursively outputs a

single prefix containing a *merge* of (modified) output actions, after which it updates its values.

In case there is input, executing the prefix

$$\left[\,\|_{i\in Extern}\, a_i(x_i)\right]$$

includes the performance of the external read actions, after which the network can continue with its updated data state.

7. An example: the wave equation

In this section we study a parallel algorithm for an approximation of a wave equation. We specify a given algorithm for this approximation in a single-output and connected network. The resulting network can be characterized as a *grid protocol.* We refrain from a formal definition of such protocols, and – as stated before – use the term for networks that can be associated with grids (processors or points of measure in some physical phenomenon, for instance a string). Finally, we present some simulation results.

7.1. The wave equation

Consider the following linear homogeneous partial differential equation:

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

This equation is known in wave mechanics as the *one-dimensional wave equation*; it describes the transversal propagation along the x-coordinate or *amplitude* $y(x,t)$ of a wave. Various wave phenomena may be modeled by this equation. One can think for instance of vibrations in a string, where it is required that the tension in the string is approximately constant. The constant c is described by $\sqrt{T/\rho}$, where T is the tension in the string and ρ the string mass per unit of length. In general, in solutions $y(x,t)$ the constant c is interpreted as the propagation velocity of the wave in transversal direction. Throughout this section we keep the string example in mind.

In order to solve the wave equation *boundary conditions* and *initial conditions* are needed. As boundary conditions we assume that $y(0,t) = y(l,t) = 0$ for $t \geqslant 0$, 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. 1.

We moreover require that we have $y(x,0)$ and $\partial y/\partial t|_{t=0}$ as given initial conditions for $0 \leqslant x \leqslant l$. From these two functions 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 $y(x,\Delta t)$ will be needed later on for initializing 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

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

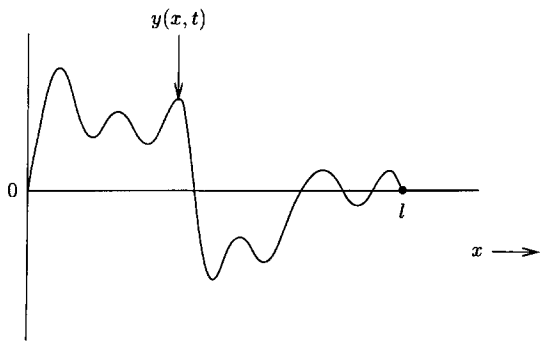


Fig. 1. Some string amplitude at time t.

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 $1 \leqslant i \leqslant N-1$, and $t \geqslant 2\Delta t$ (see e.g. [11, 21]). 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 subsequent pages we specify such a parallel algorithm based on the networks as studied in the previous section. 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 behavior of the algorithm that these three requirements are satisfied. This equation immediately follows from Theorem 5.5.

7.2. *A grid protocol modeling the wave*

In the previous section we established the following equation for the calculation of the new value of coordinate y_i:

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

This equation 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:

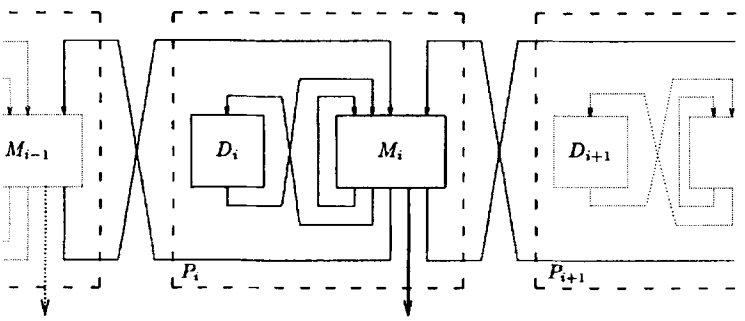
- the output of processor P_{i-1},
- the output of processor P_i (itself),
- the output of processor P_{i+1}, and
- the *previous* output of processor P_i (itself).

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 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:

$$D_i(e) = \tau_{\{c_{(D_i)}\}} \circ \partial_{\{r_{(D_i)}, s_{(D_i)}\}}(RD_i \,\|\, SD_i(e)),$$
$$RD_i = (er_{(M_i,D_i)}(v); s_{(D_i)}(v))^* \delta,$$

$$SD_i = (er_{(D_i)}(v); s_{(D_i,M_i)}(v))^*\delta,$$
$$SD_i(e) = s_{(D_i,M_i)}(e) \cdot SD_i.$$

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:

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

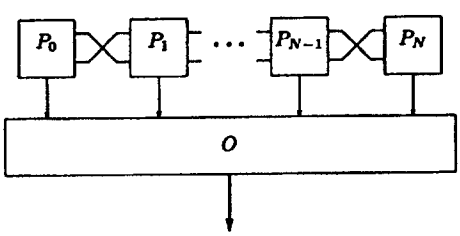
The port (M_i,O) is the actual output port of the processor, leading to an output module O.

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 $(0 \leqslant i \leqslant N)$ in the processors is sent to output module O. This module collects the computed values of all processors and bundles them in a vector. 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:

$$
\begin{aligned}
P_0(d) &= \tau_{\{c_{(P_0)}\}} \circ \partial_{\{r_{(P_0)},s_{(P_0)}\}}(R_0 \parallel S_0(d)) \\
R_0 &= (er_{(M_1,P_0)}(v); s_{(P_0)}(0))^* \delta \\
S_0 &= (er_{(P_0)}(v); (s_{(P_0,M_1)}(v) \parallel s_{(P_0,O)}(v)))^* \delta \\
S_0(d) &= (s_{(P_0,M_1)}(d) \parallel s_{(P_0,O)}(d)) \cdot S_0, \\
P_N(d) &= \tau_{\{c_{(P_N)}\}} \circ \partial_{\{r_{(P_N)},s_{(P_N)}\}}(R_N \parallel S_N(d)) \\
R_N &= (er_{(M_{N-1},P_N)}(v); s_{(P_N)}(0))^* \delta \\
S_N &= (er_{(P_N)}(v); (s_{(P_N,M_{N-1})}(v) \parallel s_{(P_N,O)}(v)))^* \delta \\
S_N(d) &= (s_{(P_N,M_{N-1})}(d) \parallel s_{(P_N,O)}(d)) \cdot S_N.
\end{aligned}
$$

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:

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

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

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

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

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

The external behavior 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 5.5, which gives the following characterization of our specification:

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

7.3. *Simulation of the grid protocol*

The early-read format that we used in the previous section has been formalized as an adaptation of the specification language μCRL [15]. See [16] for a description of this adaptation. In the same paper, a tool is described, which translates a specification in the early-read format into a specification without early reads. This makes it possible to use the simulator from the PSF Toolkit [23] which means that we are able to simulate our specification. The PSF Toolkit, to which the simulator belongs, is a set of tools designed for the specification language PSF [18, 19]. Since μCRL can be considered a small subset of PSF, an adapter was written so that the PSF Toolkit could be used for μCRL specifications as well. By removing the early reads from our specification, we make the specification suited for this adapter.

The early reads can be removed, because they form no *functional* extension to μCRL. They have been added with the purpose of simplifying the specification of grid protocols, but as argued before, it is possible to specify these protocols without early reads. As an example, we give a specification of R_i from the previous section:

$$\begin{aligned}
R_i = &((er_{(M_i,M_i)}(v_1) \parallel er_{(D_i,M_i)}(v_2) \parallel er_{(M_{i-1},M_i)}(v_3) \parallel er_{(M_{i+1},M_i)}(v_4)); \\
&s_{(M_i)}(F(v_1,v_2,v_3,v_4)))^* \delta.
\end{aligned}$$

Without early reads, a straightforward specification of R_i is the following:

$$\begin{aligned}
R_i = &\Big(\sum_{v_1}(r_{(M_i,M_i)}(v_1) \cdot \Big(\sum_{v_2}(r_{(D_i,M_i)}(v_2) \cdot \Big(\sum_{v_3}(r_{(M_{i-1},M_i)}(v_3) \\
&\cdot \sum_{v_4}(r_{(M_{i+1},M_i)}(v_4) \cdot s_{(M_i)}(F(v_1,v_2,v_3,v_4)))) \\
&+\sum_{v_4}(r_{(M_{i+1},M_i)}(v_4) \cdot \sum_{v_3}(r_{(M_{i-1},M_i)}(v_3) \cdot s_{(M_i)}(F(v_1,v_2,v_3,v_4)))))) \\
&+\sum_{v_3}(r_{(M_{i-1},M_i)}(v_3) \cdot \Big(\sum_{v_2}(r_{(D_i,M_i)}(v_2) \cdot \sum_{v_4}(r_{(M_{i+1},M_i)}(v_4) \\
&\cdot s_{(M_i)}(F(v_1,v_2,v_3,v_4)))) \\
&+\sum_{v_4}(r_{(M_{i+1},M_i)}(v_4) \cdot \sum_{v_2}(r_{(D_i,M_i)}(v_2) \cdot s_{(M_i)}(F(v_1,v_2,v_3,v_4)))))) \\
&+\sum_{v_4}(r_{(M_{i+1},M_i)}(v_4) \cdot \Big(\sum_{v_2}(r_{(D_i,M_i)}(v_2) \cdot \sum_{v_3}(r_{(M_{i-1},M_i)}(v_3) \\
&\cdot s_{(M_i)}(F(v_1,v_2,v_3,v_4))))
\end{aligned}$$

$$
\begin{aligned}
&+\sum_{v_3}(r_{(M_{i-1},M_i)}(v_3)\cdot\sum_{v_2}(r_{(D_i,M_i)}(v_2)\cdot s_{(M_i)}(F(v_1,v_2,v_3,v_4)))))))\\
&+\sum_{v_2}(r_{(D_i,M_i)}(v_2)\cdot\cdots\\
&+\sum_{v_3}(r_{(M_{i-1},M_i)}(v_3)\cdot\cdots\\
&+\sum_{v_4}(r_{(M_{i+1},M_i)}(v_4)\cdot(\sum_{v_1}(r_{(M_i,M_i)}(v_1)\cdot(\sum_{v_2}(r_{(D_i,M_i)}(v_2)\\
&\quad\cdot\sum_{v_3}(r_{(M_{i-1},M_i)}(v_3)\cdot s_{(M_i)}(F(v_1,v_2,v_3,v_4))))\\
&+\sum_{v_3}(r_{(M_{i-1},M_i)}(v_3)\cdot\sum_{v_2}(r_{(D_i,M_i)}(v_2)\cdot s_{(M_i)}(F(v_1,v_2,v_3,v_4))))))\\
&+\sum_{v_2}(r_{(D_i,M_i)}(v_2)\cdot(\sum_{v_1}(r_{(M_i,M_i)}(v_1)\cdot\sum_{v_3}(r_{(M_{i-1},M_i)}(v_3)\\
&\quad\cdot s_{(M_i)}(F(v_1,v_2,v_3,v_4))))\\
&+\sum_{v_3}(r_{(M_{i-1},M_i)}(v_3)\cdot\sum_{v_1}(r_{(M_i,M_i)}(v_1)\cdot s_{(M_i)}(F(v_1,v_2,v_3,v_4))))))\\
&+\sum_{v_3}(r_{(M_{i-1},M_i)}(v_3)\cdot(\sum_{v_1}(r_{(M_i,M_i)}(v_1)\cdot\sum_{v_2}(r_{(D_i,M_i)}(v_2)\\
&\quad\cdot s_{(M_i)}(F(v_1,v_2,v_3,v_4))))\\
&+\sum_{v_2}(r_{(D_i,M_i)}(v_2)\cdot\sum_{v_1}(r_{(M_i,M_i)}(v_1)\cdot s_{(M_i)}(F(v_1,v_2,v_3,v_4)))))))\\
&)^*\delta.
\end{aligned}
$$

Note that we do not claim that this is the shortest μCRL specification possible. In [16] a shorter, yet more tricky approach is presented. However, we think that this example shows that the early-read format is a useful syntactical extension, which allows for compact and simple specifications. Having automatically derived a conventional μCRL expression, we are now able to use it as input for the simulator.

Figs. 2–5 show different states of the simulation of module M_1. We left out encapsulation and abstraction to be able to show all actions. It is for this that we will speak of the *unsynchronized* module M_1. Each figure shows a *Choose* and a *Trace* window. The Choose window presents the possible actions that can be performed in the current state. The Trace window shows the actions that have been performed since initialization.

Fig. 2 shows the initial state M_1(R0): no actions have been executed yet. In the Choose window, one can clearly distinguish the sending and the receiving part of the module. Some initial output value R0 can be sent via the five output ports; from the four input ports arbitrary values can be read. Each read and send action has two arguments. The first argument is the port that is being used, the second argument contains the actual data (floating reals). The sums surrounding the read actions indicate that an arbitrary value can be read. Which value will be read depends on which value is sent by the other modules (or, in one case, by the same module). It can be seen that all read and send actions can be executed in arbitrary order: each of the four read and five send actions can be chosen.

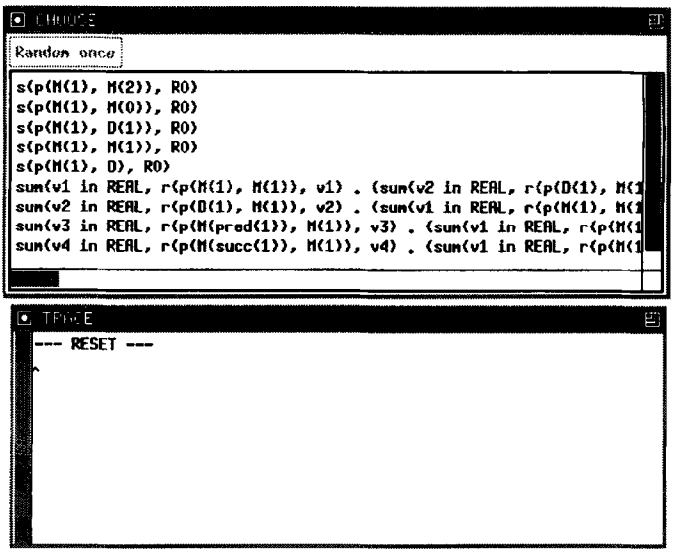


Fig. 2. M_1(R0), the initial state of (unsynchronized) module M_1 with current value R0.

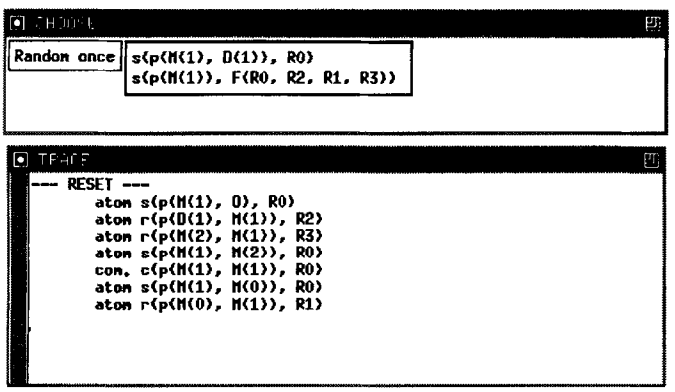


Fig. 3. Module M_1 (unsynchronized) after reading all input and sending almost all output.

Fig. 3 shows the state in which all input has been read, and the initial output value R0 has been sent to all but one output ports (the port p(M(1), D(1))). The read and send action on port p(M(1), M(1)) have synchronized into the communication action c(p(M(1), M(1)), R0). The Trace window shows that the reading and sending has been interspersed. The Choose window shows that the receiving part of the module has read the real numbers R0 to R3 and is ready to send the calculated value F(R0, R2, R1, R3) on the internal port p(M(1)) to the sending part. The sending part, however, is not yet ready to receive this value, because it still has to send the initial output value R0 to one output port (p(M(1), D(1))).

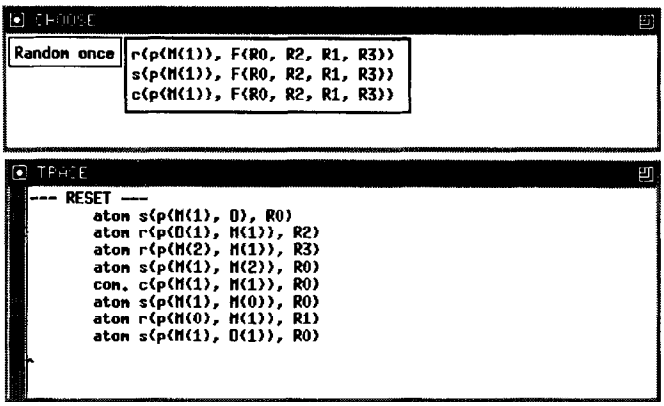


Fig. 4. Module M_1 (unsynchronized!) ready to transfer the calculated value.

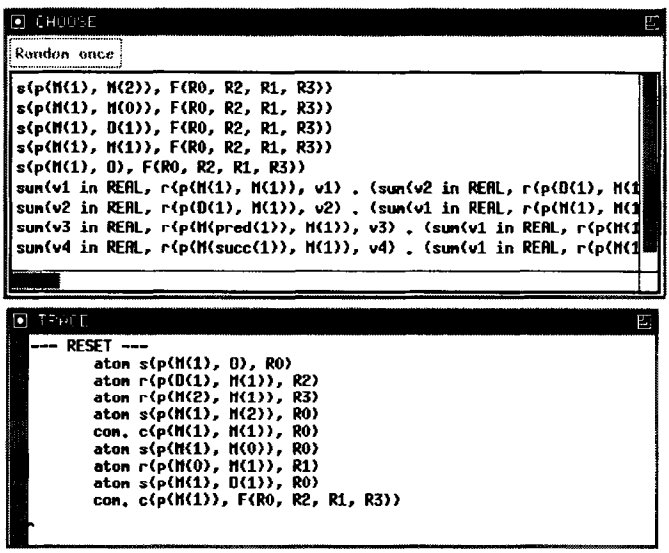


Fig. 5. M_1(F(R0, R2, R1, R3)), module M_1 (unsynchronized) with the next current value.

Fig. 4 shows the next state, in which the initial output value R0 has also been sent to the final output port. The Choose window now shows that the calculated value can be read (by the sending part), sent (by the receiving part – which was already the case), and therefore be communicated on the internal port p(M(1)). Note that the separate read and send action on this internal port are visible only because we simulate an unsynchronized version of M_1. When simulating the module with encapsulation, this communication is enforced. Separate read and send actions will then be prohibited.

The communication mentioned above has taken place in Fig. 5. The unsynchronized module has now evolved in M_1(F(R0, R2, R1, R3)), a state similar to the initial

state, as can be seen in the Choose window. The sending part of the module will now send the calculated value `F(R0, R2, R1, R3)` to its output ports, instead of the initial output value `R0`.

8. Conclusions

We hope to have shown that process prefixing and early reads allow for a short and clear notation of parallel algorithms and other concurrent phenomena. We think it is a useful extension to ACP-based specification formalisms. Of course, much work remains to be done, for example regarding extensions in the field of asynchronous networks.

Acknowledgements

We thank Jos van Wamel and Gerard Kok for the many contributions and discussions that led to the final version of this paper. The present Section 7.1 is an extract from an earlier work by these authors. We further thank Jaap Kaandorp, Matthew Poole, Peter Sloot, John Tucker and the referees for comments and discussions.

The research of J.A. Bergstra and A. Ponse was partially sponsored by Esprit Working Group 8533 NADA – New Hardware Design Methods.

References

[1] J.C.M. Baeten and J.A. Bergstra, On sequential composition, action prefixes and process prefix, *J. Formal Aspects Comput. Sci.* **6** (3) (1994) 83–98.

[2] J.C.M. Baeten, J.A. Bergstra and J.W. Klop, Conditional axioms and α/β calculus in process algebra, in: M. Wirsing, ed., *Formal Description of Programming Concepts – III, Proc. 3rd IFIP WG 2.2 Working Conf.*, Ebberup, 1986 (North-Holland, Amsterdam, 1987) 53–75.

[3] J.C.M. Baeten and W.P. Weijland. *Process Algebra*, Cambridge Tracts in Theoretical Computer Science, Vol. 18 (Cambridge Univ. Press, Cambridge, 1990).

[4] J.A. Bergstra, I. Bethke and A. Ponse, Process algebra with iteration and nesting, *Comput. J.* **37** (4) (1994) 243–258.

[5] J.A. Bergstra and J.W. Klop, The algebra of recursively defined processes and the algebra of regular processes, in: J. Paredaens, ed., *Proc. 11th ICALP*, Antwerpen, Lecture Notes in Computer Science, Vol. 172 (Springer, Berlin, 1984) 82–95; an extended version appeared in: A. Ponse, C. Verhoef and S.F.M. van Vlijmen, eds., *Algebra of Communicating Processes, Utrecht 1994*, Workshops in Computing (Springer, Berlin, 1995) 1–25.

[6] J.A. Bergstra and J.W. Klop, Algebra of communicating processes with abstraction, *Theoret. Comput. Sci.* **37** (1) (1985) 77–121.

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

[8] J.A. Bergstra and J.V. Tucker, Top-down design and the algebra of communicating processes, *Sci. Comput. Programming* **5** (2) (1985) 171–199.

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

[10] W.J. Fokkink and H. Zantema, Basic process algebra with iteration: completeness of its equational axioms, *Comput. J.* **37** (4) (1994) 259–267.

[11] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon and D. Walker, *General Techniques and Regular Problems*, Solving Problems on Concurrent Processors, Vol. 1 (Prentice-Hall, Englewood Cliffs, NJ, 1988).

[12] J.F. Groote and H. Korver, A correctness proof of the bakery protocol in μCRL, in: A. Ponse, C. Verhoef and S.F.M. van Vlijmen, eds., *Algebra of Communicating Processes, Utrecht 1994*, Workshops in Computing (Springer, Berlin, 1995) 63–86.

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

[14] 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, eds., *Proc. Internat. Workshop on Semantics of Specification Languages*, Workshops in Computing (Springer, Berlin, 1994) 232–251.

[15] J.F. Groote and A. Ponse, The syntax and semantics of μCRL, in: A. Ponse, C. Verhoef and S.F.M. van Vlijmen, eds., *Algebra of Communicating Processes, Utrecht 1994*, Workshops in Computing (Springer, Berlin, 1995) 26–62.

[16] J.A. Hillebrand, A small language for the specification of grid protocols, Technical Report p9608, Programming Research Group, University of Amsterdam, 1996. Also appeared in: Experiments in specification re-engineering, PhD. thesis, University of Amsterdam, 1996.

[17] S.C. Kleene, Representation of events in nerve nets and finite automata, in: *Automata Studies* (Princeton Univ. Press, Princeton, NJ, 1956) 3–41.

[18] S. Mauw and G.J. Veltink, A process specification formalism, *Fund. Inform.* **XIII** (1990) 85–139.

[19] S. Mauw and G.J. Veltink eds., *Algebraic Specification of Communication Protocols*, Cambridge Tracts in Theoretical Computer Science, Vol. 36 (Cambridge Univ. Press, Cambridge, 1993).

[20] R. Milner, *Communication and Concurrency* (Prentice-Hall, Englewood Cliffs, NJ, 1989).

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

[22] B.C. Thompson and J.V. Tucker, Equational specification of synchronous concurrent algorithms and architectures (2nd ed.), Report CSR 15-94, University of Wales, Swansea, 1994.

[23] G.J. Veltink, The PSF toolkit, *Comput. Networks ISDN Systems* **25** (1993) 875–898.