Grid Protocols based on Synchronous Communication: Specification and Correctness

Jan A. Bergstra[1,2] Joris Hillebrand[1] 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` - `joris@fwi.uva.nl` - `alban@fwi.uva.nl`

Abstract

We provide a short notation for processes with parallel inputs and outputs. With this specification format, a certain class of networks, comprising synchronous grid protocols, can be easily specified and analyzed. We illustrate this by an example on the approximation of a one-dimensional wave equation.

1 Introduction

In [BB94], Baeten and Bergstra proposed axiom systems for *action prefixes* and *process prefixes*, in the context of the process algebra ACP (Algebra of Communicating Processes [BK84, BK85, BW90]). Process prefixes form a generalization of action prefixes, which in its turn provide a means for binding variables as modeled in the value-passing calculus of the related formalism CCS (Calculus of Communicating Systems, Milner [Mil89].)

Due to the intrinsic difficulties that go with variable binding, expressing parallel inputs is not straightforward in most ACP-based approaches. As an example, let P be a process that is able to read two naturals along ports 1 and 2. If values n and m are read, respectively, the process evolves into $Q(n,m)$. In case the particular order of reading is immaterial (or *should be* irrelevant), we have a typical case of "parallel inputs". The process P can then be specified as

$$\begin{aligned} P \;=\; & \sum\Big(v:\mathbb{N},\ r_1(v)\cdot\sum\big(w:\mathbb{N},\ r_2(w)\cdot Q(v,w)\big)\Big)\ + \\ & \sum\Big(w:\mathbb{N},\ r_2(w)\cdot\sum\big(v:\mathbb{N},\ r_1(v)\cdot Q(v,w)\big)\Big) \end{aligned}$$

where the reading is interleaved: either first along port 1 and then along port 2 (first summand), or vice versa (second summand). So the order in which binding takes place is not relevant. The variable binding properties of the $\sum$-operator, however, are straightforward: its scope is given by the brackets in $\sum(v:\mathbb{N},\ldots)$. Consequently, there is no way of expressing P using the parallel operator $\|$. With *early read* actions this problem is easily solved:

$$(er_1(v)\ \|\ er_2(w)); Q(v,w)$$

fully expresses P. We think it is obvious that parallel inputs may form an important technical expedient in the study of various kinds of processes.

The purpose of this paper is to introduce this specification format in a general setting, and to illustrate it by a specific type of protocols. After a brief introduction to the axiom system $\mathrm{ACP}^\tau(A,\gamma)$ and iteration, we present $\mathrm{ACP}^\tau_{er}(A,\gamma)$, which stands for $\mathrm{ACP}^\tau(A,\gamma)$ with process prefixes and early read actions. Then, in Section 3, we define *modules*. Modules may be regarded elementary processors of data, with a memory for one data element. A component-wise specification of such a module is provided, and an elegant expression for its *external* behaviour is derived. Modules can be composed into *networks*, of which a simple type occurs in this paper: we shall speak of a *grid protocol* whenever the way all modules are connected is typical for the process. For a certain class of networks, the equation characterizing external behaviour follows from a general theorem. This equation in turn implies the *correctness* of the specification.

We illustrate our specification format for grid protocols in Section 4 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 *network* of modules, from which its correctness follows. We pay some attention to the elimination of process prefixes and early read actions in the specification (cf. the example above), and to simulation issues. Section 5, containing some conclusions, ends the paper.

Related work. Our modeling of modules and grid protocols is very much based on the work done on Synchronous Concurrent Algorithms (SCAs) in Swansea (Tucker and Thompson, [TT94]). A particular reason to follow the Swansea approach is given by the following citation: "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 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.

In [BB94], the authors develop a different approach to process prefixing. They circumvent the use of typed *variables* in value-transmission, and the related problem of α-conversion. A price to be paid is the restriction to a *finite data type*.

In the simultaneous paper [BKPW] we set out a rigorous proof theory for early reads and process prefixes with (globally) typed variables. This approach is related — and connected — to μCRL, a formal framework for the specification and verification of processes with data [GP95, GP91c, GP94b].

Finally, in [Hil95] 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 ACP with Iteration and Process Prefixing

In this section we introduce the syntax that this paper is about. First we recall the well-known system $\mathrm{ACP}^\tau(A,\gamma)$ and shortly introduce iteration. Then we discuss the way in which data and value-transmission are specified. Finally, we introduce an operator and axioms for process prefixing.

2.1 Process Algebra with Iteration

We start off with the axioms of $\mathrm{ACP}^\tau(A,\gamma)$, displayed in Table 1. The idea is that we have a set A of atomic actions and a (partial) binary, commutative communication function γ on A. Furthermore,

(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	$=$	δ				
(CF1)	$a \mid b$	$=$	$\gamma(a,b)$ if $\gamma(a,b)\downarrow$				
(CF2)	$a \mid b$	$=$	δ otherwise				
(CM1)	$x \parallel y$	$=$	$x \mathbin{\|\!\!\lfloor} y + y \mathbin{\|\!\!\lfloor} x + x \mid y$				
(CM2)	$a \mathbin{\|\!\!\lfloor} x$	$=$	ax				
(CM3)	$ax \mathbin{\|\!\!\lfloor} y$	$=$	$a(x \parallel y)$				
(CM4)	$(x+y) \mathbin{\|\!\!\lfloor} z$	$=$	$x \mathbin{\|\!\!\lfloor} z + y \mathbin{\|\!\!\lfloor} z$				
(CM5)	$ax \mid b$	$=$	$(a \mid b)x$				
(CM6)	$a \mid bx$	$=$	$(a \mid b)x$				
(CM7)	$ax \mid by$	$=$	$(a \mid b)(x \parallel y)$				
(CM8)	$(x+y) \mid z$	$=$	$x \mid z + y \mid z$				
(CM9)	$x \mid (y+z)$	$=$	$x \mid y + x \mid z$				
(D1)	$\partial_H(a)$	$=$	a if $a \notin H$	(TI1)	$\tau_I(a)$	$=$	a if $a \notin I$
(D2)	$\partial_H(a)$	$=$	δ if $a \in H$	(TI2)	$\tau_I(a)$	$=$	τ if $a \in I$
(D3)	$\partial_H(x+y)$	$=$	$\partial_H(x)+\partial_H(y)$	(TI3)	$\tau_I(x+y)$	$=$	$\tau_I(x)+\tau_I(y)$
(D4)	$\partial_H(xy)$	$=$	$\partial_H(x)\cdot\partial_H(y)$	(TI4)	$\tau_I(xy)$	$=$	$\tau_I(x)\cdot\tau_I(y)$

Table 1: The axioms of $\mathrm{ACP}^\tau(A,\gamma)$ where $a,b \in A_{\delta,\tau}$, $H, I \subseteq A$.

there are operations for, among others, alternative composition ($+$), sequential composition ($\cdot$), and parallel composition ($\parallel$). Using these operations, larger process expressions can be built from atomic actions, and can be manipulated according to the axioms of $\mathrm{ACP}^\tau(A,\gamma)$. We mostly suppress the $\cdot$ in process expressions, and brackets according to the following precedences: $\cdot > \{\parallel, \mathbin{\|\!\!\lfloor}, \mid\} > +$. Note that $+$ and $\cdot$ are associative. We further assume associativity of $\parallel$ and $\mid$, also known as Standard Concurrency [BT84]. For a detailed introduction to $\mathrm{ACP}^\tau(A,\gamma)$, and an overview we refer to [BW90].

In order to describe iterative processes we shall use the (binary) Kleene star [Kle56, BBP94], 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. If x is a terminating process, $x^*\delta$ is the process that repeatedly executes x.

Remark 2.1. For the interested reader, we elaborate a little on the way one can reason with iterative processes. Besides the equational axioms introduced before, there is the principle RSP* (Recursive Specification Principle for *-processes):

$$(\mathrm{RSP}^*) \qquad x = yx + z \quad \Longrightarrow \quad x = y^* z.$$

For example, one easily proves with RSP* that $(a + b\delta)^* c = a^*(b\delta + c)$:

$$\begin{aligned}(a+b\delta)^* c &= (a+b\delta)((a+b\delta)^* c) + c \\ &= a((a+b\delta)^* c) + b\delta((a+b\delta)^* c) + c \\ &= a((a+b\delta)^* c) + b\delta + c.\end{aligned}$$

So $(a+b\delta)^* c$ satisfies the equation $x = ax + (b\delta + c)$. With RSP* it follows that $(a+b\delta)^* c = a^*(b\delta + c)$. Some caution is needed in a setting with τ because RSP* may induce undesirable equivalences. For example, $\tau \cdot x = \tau(\tau \cdot x) + \delta$, but it is very undesirable that $\tau \cdot x = \tau^* \delta$. The conditions under which RSP* is acceptable have been described in [BBP94a] (these refer to certain forms of guardedness).
End Remark.

2.2 Data and Value-Transmission

In order to reason about processes that manipulate data, we need some minimal assumptions about the data involved (such as decidability). We adopt a simple specification paradigm for data and actions parameterized with data, originating from μCRL [GP95]. Let for instance a be typed as an action that can carry values of $\mathbb{N}$ (the natural numbers) and of $\mathbb{N} \times \mathbb{N}$. So $a(0), a(1), ..., a(0, 0), ...$ are considered actions. We use data in process expressions in two ways: in *data-parametric sums*, and in *communications*.

An example of a data-parametric sum is the expression $\sum(v : \mathbb{N}, a(v))$, denoting a process that for any value n of $\mathbb{N}$ can once perform $a(n)$ after which it is terminated. A typical use of this construct is

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

which characterizes the infinite summation $a(0)a(0, 0) + a(1)a(1, 2) + a(2)a(2, 4) + \cdots$. 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 [GP91c, GP94b]. 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 2. 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 2 are the only communications defined; this means in particular that the *handshaking* paradigm is satisfied [BT84]. With help of send-read communication and the

Table 2: Send-read communication for value-transmission.

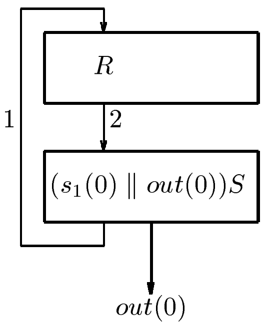
$$r_i(t_0) \mid s_j(t_1) = \begin{cases} c_i(t_0) & \text{if } i = j \text{ and } t_0 = t_1 \\ \delta & \text{otherwise.} \end{cases}$$

operations ∂_H and τ_I one can easily model value-transmission. We illustrate this by an example on a simple, one-module network. In the sets of actions H and I, we further only mention the identifiers r_i, s_i and c_i (conform to the definition of μCRL).

Example 2.2. Consider the following two processes, abbreviated by R and S:

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

Let $H=\{r_1,s_1,r_2,s_2\}$. We first analyze the behaviour of $\partial_H(R\parallel(s_1(0)\parallel out(0))S)$. Note that by the ∂_H, single r_i and s_i actions cannot occur, and are thus enforced to communicate. Actions $out(j)$ cannot be involved in a communication. In a picture:



With the axioms of ACP$^\tau(A,\gamma)$ and (BKS1) it easily follows that

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

In the first summand of the right-hand side, we see the value-transmission of 0 along channel 1: the action $s_1(0)$ can only communicate with the summand $r_1(0)\cdot s_2(0)$ of R. In this way we derive:

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

Let $I=\{c_2\}$ and let $P(x)=\tau\cdot\tau_I\circ\partial_H(R\parallel(s_1(x)\parallel out(x))S)$. It follows that

$$P(x)=\tau\cdot out(x)\cdot P(x+1)$$

expressing that $P(n)$ outputs the infinite stream $\tau\cdot out(n)\cdot out(n+1)\cdot out(n+2)\cdot\ \cdots$.
End example.

2.3 Process Prefixing

For the sequel, we assume a decidable data type D (with only total functions over D), variables $v,w,v_1,v_2,v_3,\ldots$ of type D, and values (closed terms) $d,e,\ldots$ of type D. We consider process prefixing and early read actions $er_i(v)$, with i a channel or port identifier. The early read axiom scheme, parameterized with data type D, is then

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

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

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

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

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

Let A_{er} be the extension of A (the set of atomic actions) with early read actions for any declared action $r_i : D_1 \times ... \times D_n$. Axioms for process prefixing are given in Table 3. The axiom PP4 is considered to be parameterized with the typing of the r_i action. As an example, an early read over *pairs of values* gives rise to two $\sum$-applications (which commute [GP91c, GP94b]), e.g., for $r_k : D \times \mathbb{N}$ we obtain:

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

Table 3: Process prefixing, $a \in A$.

(PP1)	$\delta; x$	$=$	δ	|	(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)	$(x \cdot y); z$	$=$	$x; (y; z)$

The process $P(0)$ defined in Example 2.2 can now be specified by

$$\tau \cdot \tau_I \circ \partial_H\Big(((er_1(v); s_2(v+1))^* \delta) \;\Big\|\; (s_1(0) \parallel out(0)) \cdot (er_2(v); (s_1(v) \parallel out(v))^* \delta) \Big)$$

with v a variable of type $\mathbb{N}$. Note that for the er actions we use globally typed variables.

3 Modules and Networks

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.

3.1 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 $i_1, ..., i_n$ and output channels $o_1, ..., o_m$ 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 $o_1, ..., o_m$). These two parts communicate along some channel i:

$$M_i(d) = \tau_{\{c_i\}} \circ \partial_{\{r_i,s_i\}}(Rec_i \parallel Send_i(d))$$

$$\begin{aligned} Rec_i &= ((er_{i_1}(v_1) \parallel ... \parallel er_{i_n}(v_n)); s_i(F_i(v_1, ..., v_n)))^* \delta \\ Send_i(d) &= (s_{o_1}(d) \parallel ... \parallel s_{o_m}(d)) \cdot Send_i \\ Send_i &= (er_i(v); (s_{o_1}(v) \parallel ... \parallel s_{o_m}(v)))^* \delta \end{aligned}$$

where the channel i is internal. 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. It follows that $M_i(d)$ has a process prefix

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

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 communication along channel i, renamed into the silent action τ), and the next process prefix is ready to be performed:

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

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

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

so $M_i(d) = Rec_i \parallel_i Send_i(d)$.

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.

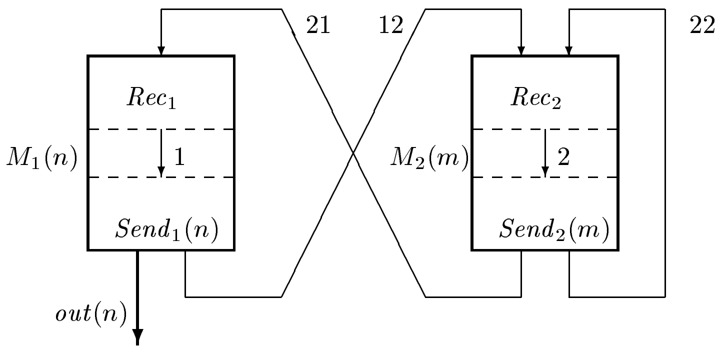
3.2 Networks

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 restrict attention to networks that also satisfy the following two constraints:

1. Networks that are *single-output*: there is exactly one external (output-) action, which for convenience will be referred to as $out(..)$. So the external behaviour of a single-output network is a sequence of $out(..)$ actions.
2. Networks that are *output-connected*: from each module one can traverse along the network's channels to its 'output-module', i.e., the module containing the $out(..)$ action.

In this way, single read and send-actions cannot occur: synchronizations (communications between modules) are thus enforced, after which they are hidden (renamed into the silent action τ). Below we give an example for computing a Fibonacci sequence using a network of two modules M_1 and M_2. This example also illustrates the reason for computing a next value in a module only after all the current input *and output* actions have been executed: the latter may have to communicate with some of the module's input actions. We call this phenomenon *feedback*. Note that the process $P(x)$ defined in Example 2.2 exemplifies the most simple network that is single-output and output-connected.

Example 3.1. *Fibonacci Network.* Consider the following network in which all data to be transmitted are or 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 $M_2(m)$:

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

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

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

computes the ordinary Fibonacci sequence $1, 1, 2, 3, 5, 8, ...$ as the values of its consecutive *out*-actions:

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

A different characterization is given by the equation

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

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 be easily grasped from the picture above; its correctness follows from Theorem 3.2 below.
End example.

For finitary networks that are single-output and output-connected, we have the following result (for a proof see [BKPW]):

Theorem 3.2. *Let $n \geq 1$, $\vec{v} = v_1, ..., v_n$ be a collection of typed variables, and let*

$$N(\vec{v}) = \tau_I \circ \partial_H(M_1(v_1) \parallel M_2(v_2) \parallel ... \parallel M_n(v_n))$$

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

$$\tau \cdot N(\vec{v}) = \tau \cdot out(v_1) \cdot N(F_1(\vec{v_1}), ..., F_n(\vec{v_n}))$$

where F_i *is the value-function of module* M_i*, and if* M_i *reads values from modules* $M_{i_1}, ..., M_{i_k}$*, then* $\vec{v_i} = v_{i_1}, ..., v_{i_k}$.

4 An Example: the Wave Equation

In this section we study a parallel algorithm for an approximation of the wave equation. We specify a given algorithm for this approximation in a single-output and output-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.

4.1 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 $t \geq 0$, 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 Figure 1.

We moreover require that we have $y(x,0)$ and $\partial y/\partial t|_{t=0}$ as given initial conditions for $0 \leq x \leq l$. 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

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

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.

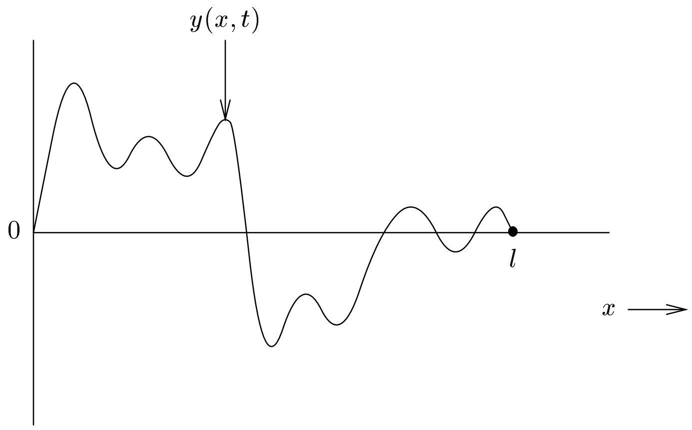


Figure 1: Some string amplitude at time t.

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 behaviour of the algorithm that these three requirements are satisfied. This equation immediately follows from Theorem 3.2.

4.2 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},

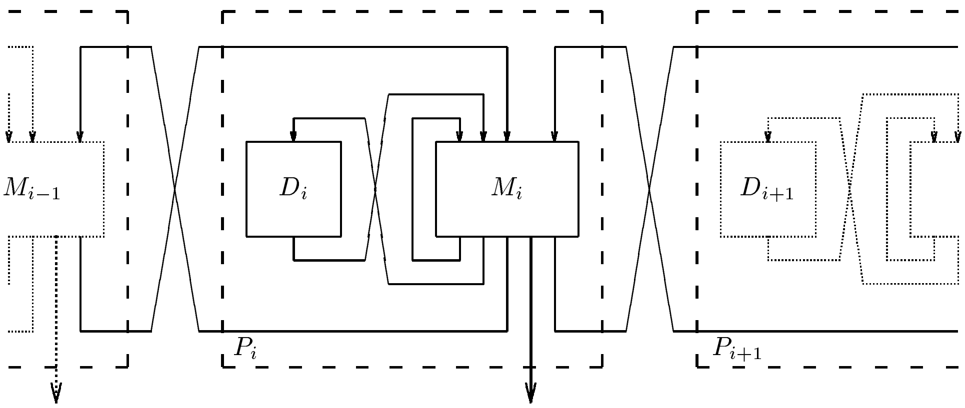
- the output of processor P_i (itself),
- the output of processor P_{i+1}, and
- the *previous* output of processor P_i (itself).

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 keeping 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:

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

Here, $er_{(M_i,D_i)}(v)$ and $s_{(M_i,D_i)}(v)$ stand for an early read or a send action on the port 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:

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

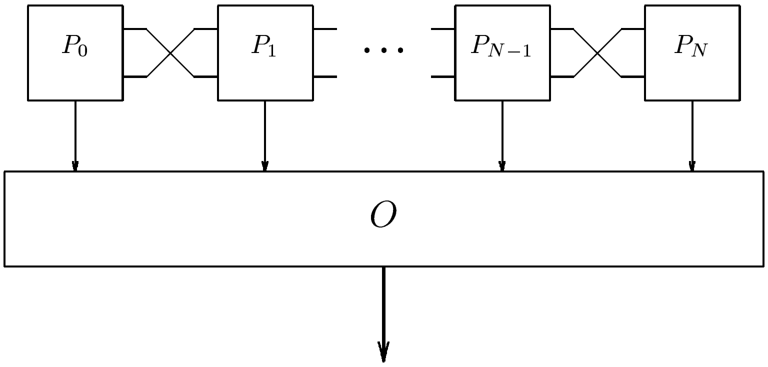
The port (M_i, O) is the actual output port of the processor, leading to an output module O.

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 $(0 \le i \le N)$ 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:

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

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

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:

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

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

$$\text{WAVE} = \tau_{\{c_p\}} \circ \partial_{\{r_p,s_p\}}(O(\vec{y}(0)) \parallel [\mathop{\parallel}_{0 \le i \le N} P_i(y_i(\Delta t), y_i(0))]\ ),$$

with $y_i(0), y_i(\Delta t)$ $(i = 0 \ldots N)$ arbitrary initial values, and p an element of 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

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

This follows from Theorem 3.2, which gives the following characterization of our specification:

$$\tau \cdot \tau_{\{c_p\}} \circ \partial_{\{r_p,s_p\}}(O(\vec{y}(k \cdot \Delta t)) \parallel [\mathop{\parallel}_{0 \le i \le N} P_i(y_i((k+1)\cdot \Delta t), y_i(k \cdot \Delta t))]\ ) \ =$$

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

4.3 Simulation of the Grid Protocol

The early-read format that we used in the previous section has been formalized as an extension to the specification language μCRL [GP95]. See [Hil95] for a description of this extension. 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 [Vel93] 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 [MV90, MV93]. 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 it is possible to specify these protocols without early reads. As an example, we give the specification of R_i from the previous section:

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

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

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

Figures 2 to 5 show different states of the simulation of module M_1. We left out encapsulation to be able to show all actions. 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.

Figure 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 (real numbers). 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.

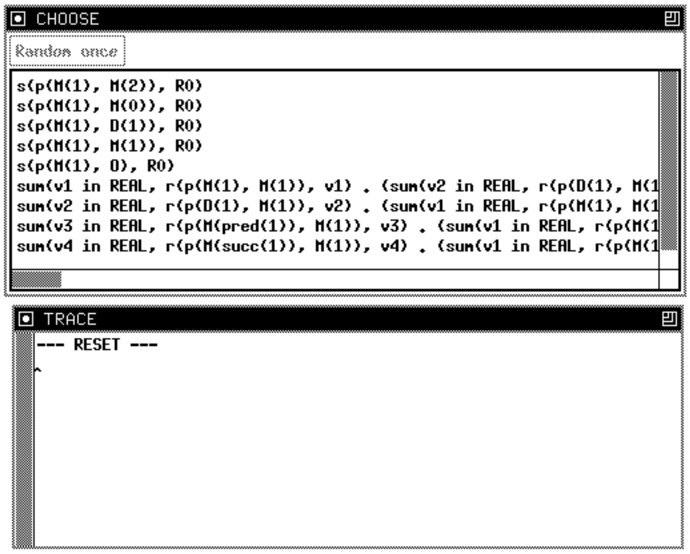


Figure 2: M_1(R0), the initial state of module M_1.

Figure 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))).

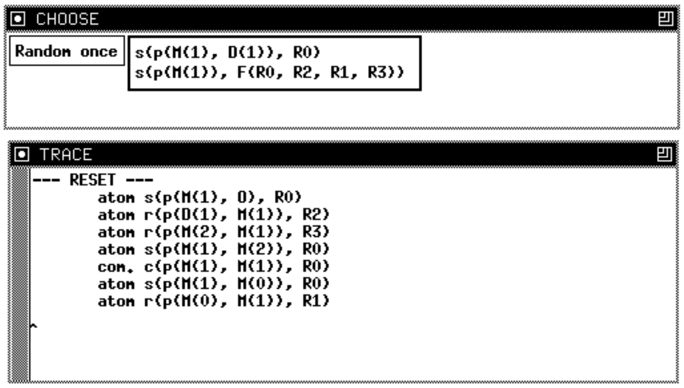


Figure 3: Module M_1 after reading all input and sending almost all output.

Figure 4 shows the next state, in which the initial output value `R0` has also be 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))`. When simulating the module with encapsulation, this communication is enforced. Separate read and send actions will then be prohibited.

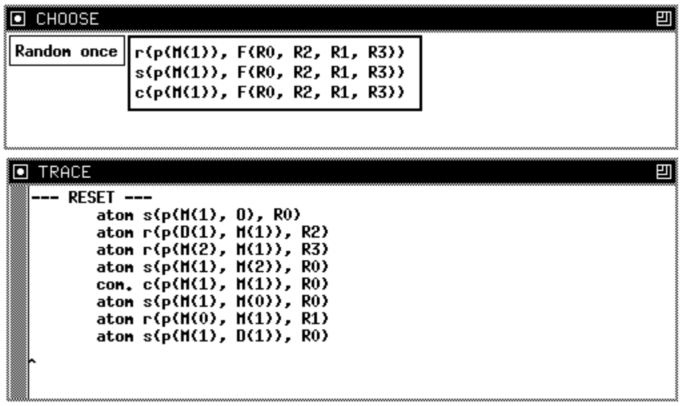


Figure 4: Module M_1 ready to transfer the calculated value.

The communication mentioned above has taken place in Figure 5. The 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`.

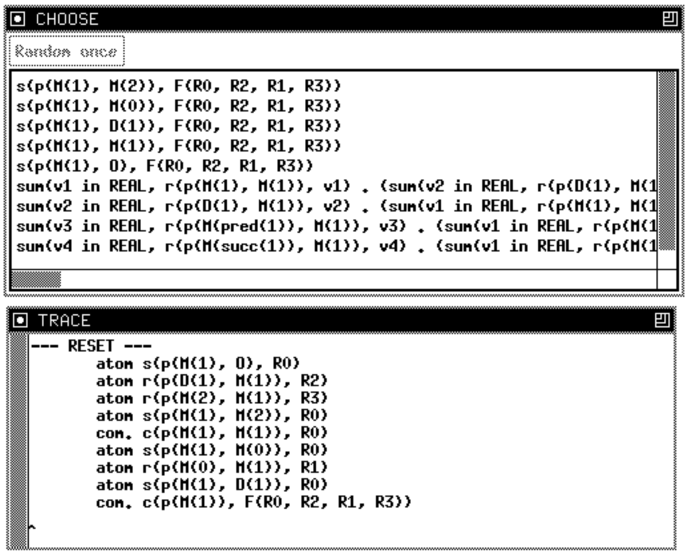


Figure 5: M_1(`F(R0, R2, R1, R3)`), module M_1 back in a state similar to the initial state.

5 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.

In this paper we paid little attention to formal details, such as the necessity of globally declared (typed) data-variables. These are considered extensively in the simultaneous paper [BKPW]. Typical for this phenomenon is that one has to manipulate expressions containing such variables (cf. the proof theory for μCRL, [GP91c, GP94b]). As an example, consider

$$(er_1(v) \parallel a(v)); b(v)$$

with a, b actions that are parameterized over D. This expression reduces to two summands, of which one is closed, and the other contains a free data-variable:

$$\begin{aligned}
(er_1(v) \parallel a(v)); b(v) &= (er_1(v) \cdot a(v) + a(v) \cdot er_1(v)); b(v) \\
&= er_1(v) \cdot a(v); b(v) + (a(v) \cdot er_1(v)); b(v) \\
&= er_1(v); (a(v) \cdot b(v)) + a(v); (er_1(v); b(v)) \\
&= \sum(v : D, r_1(v) \cdot a(v) \cdot b(v)) + a(v) \cdot \sum(v : D, r_1(v) \cdot b(v)).
\end{aligned}$$

Exchanging $a(v)$ for $er_2(v)$ (having the same variable), a kind of non-deterministic reading is modeled (without the occurrence of any 'free data-variables'):

$$(er_1(v) \parallel er_2(v)); b(v).$$

The value that is read last, is bounded; the first value-transmission has no binding effect in $b(v)$.

Acknowledgements. We like to thank our co-authors Gerard Kok and Jos van Wamel of [BKPW] for many inspiring discussions, and for contributing to the work described here.

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.

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

[BBP94] J.A. Bergstra, I. Bethke, and A. Ponse. Process algebra with iteration and nesting. *Computer Journal*, 37(4):243–258, 1994.

[BBP94a] J.A. Bergstra, I. Bethke, and A. Ponse. Process algebra with combinators. In E. Börger, Y. Gurevich, and K. Meinke, editors, *Proceedings CSL '93, Swansea*, volume 832 of *Lecture Notes in Computer Science*, pages 36–65. Springer-Verlag, 1994.

[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.

[BKPW] J.A. Bergstra, G. Kok, A. Ponse, and J.J. van Wamel. Grid Protocols Based on Synchronous Communication. Technical Report, Programming Research Group, University of Amsterdam, to appear.

[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.

[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.

[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.

[Hil95] J.A. Hillebrand. μCRL with Early Reads (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.

[MV90] S. Mauw and G.J. Veltink. A process specification formalism. *Fundamenta Informaticae*, XIII:85–139, 1990.

[MV93] S. Mauw and G.J. Veltink, editors. *Algebraic Specification of Communication Protocols.* Cambridge Tracts in Theoretical Computer Science 36. Cambridge University Press, 1993.

[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.

[Vel93] G.J. Veltink. The PSF toolkit. *Computer Networks and ISDN Systems*, 25:875–898, 1993.