Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

J.A. Bergstra, J.W. Klop, J.V. Tucker

Process algebra with asynchronous communication mechanisms

Department of Computer Science Report CS-R8410 June

PROCESS ALGEBRA WITH ASYNCHRONOUS COMMUNICATION MECHANISMS

J.A. BERGSTRA, J.W. KLOP
Centre for Mathematics and Computer Science, Amsterdam

J.V. TUCKER
University of Leeds

Algebraic specifications are given for a stimulus-response mechanism, for asynchronous non-order-preserving send and receive actions and for asynchronous order-preserving send and receive statements.

69F11, 69F12, 69F32, 69F43
1980 MATHEMATICS SUBJECT CLASSIFICATION: 68B10, 68C01, 68D25, 68F20.
1982 CR CATEGORIES: F.1.1, F.1.2, F.3.2, F.4.3.
KEY WORDS & PHRASES: concurrency, communicating processes, asynchronous communication, stimulus-response mechanism.
NOTE: This report will be submitted for publication elsewhere.

Report CS-R8410
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

INTRODUCTION

The present paper is part of a series of reports on process algebra which includes [6,7]. In [7], the starting point of this series, axiom systems PA and ACP are defined, the first describing process algebra for 'free merge', that is process co-operation consisting of mere interleaving of atomic actions without communication, the second (ACP) for Algebra of Communicating Processes describing process co-operation with communication.

This paper can be read independently though knowledge of the part of [7] concerning PA and ACP may be helpful. We also refer to [7] for a discussion of related approaches to the algebraic theory of concurrency, such as Milner's CCS [23] and SCCS [24].

The communication in ACP is, like in CCS, *synchronous* since it is based on action sharing, i.e. the simultaneous execution of so-called communication actions. In the present paper we take up the issue of *asynchronous* communication which is a mechanism of interest in programming languages such as CHILL [9].

Although we will be more explicit below, a word of clarification as regards terminology may be in order here: while the *communication* in CCS or ACP is *synchronous*, the *co-operation* between processes in these systems is *asynchronous*, as no global clock is present like in SCCS.

It is possible to define the asynchronous communication mechanisms as introduced below in terms of synchronous communication as in CCS or ACP. However, such an implementation of asynchronous communication by synchronous communication would be more involved than the syntax and corresponding axiomatisations which are presented below. These axiomatisations concern

(i) *mail via a queue-like channel*
(ii) *mail via a bag-like channel*
(iii) *a causality or stimulus-response mechanism.*

The contents of this paper are as follows:
Section 1 introduces the axiom system PA_δ describing the free merge of processes; δ is a constant for deadlock or failure. This is the axiom system underlying the three axiom systems for (i)-(iii) above. The system PA_δ was introduced first in [7].
Section 2 (*Co-operation and communication*) fixes our terminology, and attempts

a classification according to this terminology of various communication formats as in CCS, CSP, MEIJE, SCCS, CHILL, etc.
Section 3 presents the syntax and axioms for (i) and (ii) above.
Section 4 (*Causality*) introduces an axiomatisation for the mechanism of (iii).

1. PRELIMINARIES: PA_δ

As a point of departure of this exposition let us consider the following axiom system PA_δ (see Table 1 below). A is a finite set of atomic actions, $\delta \in A$ represents deadlock or failure. There are four operators on processes:

$+$	*alternative composition (sum)*
$\cdot$	*sequential composition (product)*
$\|\|$	*parallel composition (merge)*
$\|\!\lfloor$	*left-merge*

The functionality of each of these binary operators is $P \times P \rightarrow P$, where P is the class of processes. All atomic actions are processes themselves.

The axiom system PA_δ (see [7] for PA; the axioms for δ are also in [7]) is this:

PA_δ

$x + y = y + x$	A1
$(x + y) + z = x + (y + z)$	A2
$x + x = x$	A3
$x(y + z) = xy + xz$	A4
$(xy)z = x(yz)$	A5
$x + \delta = x$	A6
$\delta x = \delta$	A7
$x \| y = x \|\!\lfloor y + y \|\!\lfloor x$	M1
$a \|\!\lfloor x = ax$	M2
$(ax) \|\!\lfloor y = a(x \| y)$	M3
$(x + y) \|\!\lfloor z = x \|\!\lfloor z + y \|\!\lfloor z$	M4

Table 1.

Here x,y,z range over P and 'a' ranges over A. Products $x \cdot y$ are frequently written as xy. Let Σ_{PA_δ} be the signature of PA_δ:

$$\Sigma_{PA_\delta} \left| \begin{array}{lll} \$: & P & \textit{(sorts)} \\ \mathbb{F}: & +: P \times P \to P & \textit{(functions)} \\ & \cdot: P \times P \to P & \\ & \|: P \times P \to P & \\ & \lfloor\!\lfloor: P \times P \to P & \\ ¢: & a \in P \text{ (for all } a \in A) & \textit{(constants)} \end{array} \right.$$

There are at least two ways to consider PA_δ:

(i) As an *initial algebra specification* in the sense of ADJ [1] for the structure

$$I(PA_\delta) = T_I(\Sigma_{PA_\delta}, PA_\delta)$$

of finite processes.

(ii) As an *axiomatisation of process algebras*, i.e. as a definition of the class

$$Alg(\Sigma_{PA_\delta}, PA_\delta).$$

Let $\mathcal{P} \models PA_\delta$ be any process algebra (in the sense of PA_δ). For $p \in \mathcal{P}$ and $\alpha \in A^* \cup A^\omega$, the set of finite or infinite sequences of actions from A, we will define what it means for α to be a *trace* of p:

Definition. (i) If $\alpha = a_1 * a_2 * \ldots * a_n$, where $a_i \in A$ $(i = 1,\ldots,n)$ and * denotes concatenation, then α is a trace of p if there are $p_1,\ldots,p_n$, $q_1,\ldots,q_n,q_{n+1}$ $\in \mathcal{P}$ such that

$$p = p_1, \; p_i = a_i p_{i+1} + q_i \quad (i = 1,\ldots,n-1)$$
$$p_n = a_n + q_n$$

(ii) If $\alpha = a_1 * a_2 * \ldots$ then we call α a trace of p if there are p_i, q_i such that

$$p = p_1, \; p_i = a_i p_{i+1} + q_i \quad (i \geq 1).$$

If $p \in \mathcal{P}$ has an infinite trace, it is an *infinite* process; otherwise it is *finite*. Clearly, the initial algebra $I(PA_\delta)$ contains finite processes only.

There are various ways to construct process algebras that contain infinite processes. The synchronisation trees (modulo observational equivalence or bisimulation) from Milner [23] (see also Winskel [31]) constitute such a model if one considers the degenerate case of absence of synchronisation primitives. In De Bakker & Zucker [3,4] a topological construction is given via metric spaces, and in Bergstra & Klop [7] an algebraic construction using projective limits. Bergstra, Klop & Tucker [8] describes a direct algebraic construction by means of adjoining solutions of suitable fixed point equations.

One may imagine many more model constructions; therefore we feel that process algebra constitutes an approach which is both algebraic and axiomatic.

2. CO-OPERATION AND COMMUNICATION

From an intuitive point of view processes are configurations of atomic actions. Execution of a process works as follows: choose a first action, perform it, then choose a second action possible after the first one and perform it, and so on. *Co-operation* of processes p and q denotes their parallel execution. Two regimes of co-operation can be distinguished:

- *synchronous co-operation,*

  for instance in SCCS [14,24], ASP [7], MEIJE [2,28]: the regime of synchronous co-operation allows p and q to be executed in parallel with the same speed and timed on the same clock.

- *asynchronous co-operation,*

  for instance in CSP [15-17], CCS [23], ACP [7] and PA_δ above. Asynchronous co-operation allows p and q to proceed with their own speed and timed by independent clocks only restricted by possible mutual interactions (for CSP, CCS, ACP; in PA_δ there are no interactions).

With *communication* we denote interaction between atomic actions of processes. Again two regimes can be distinguished:

- *synchronous communication,*

  for instance in CSP, CCS, Ada: communication between actions a and b can take place only if both are performed simultaneously. This type of communication is often called hand shaking.

- *asynchronous communication,*

  for instance in CHILL [9]: send and receive statements. Communication between axtions a and b is consistent with b being performed after a.

Combining the above regimes one arrives at four possible combinations which can be used to roughly classify theoretical models of concurrency:

(AS) *asynchronous co-operation + synchronous communication:*

CCS, CSP, Ada, Petri nets, ACP, uniform processes of [3,4].

(SS) *synchronous co-operation + ~~a~~synchronous communication:*

SCCS, MEIJE, ASP, ASCCS.

(SA) *synchronous co-operation + asynchronous communication:*

no example known to us.

(AA) *asynchronous co-operation + asynchronous communication:*

CHILL, data flow networks, restoring circuit logic.

Remark. (i) It might be puzzling why ASCCS, which gives according to Milner [24] a framework for 'asynchronous processes', is classified under (SS). The reason is that it is a subcalculus of SCCS, hence also employs synchronous co-operation and synchronous communication - even though asynchronously co-operating processes may be *interpreted* in ASCCS.
(ii) Restoring circuit logic (see [13, 27, 29]) is intended to describe the behaviour of circuits regardless of delays in the connecting wires. This delay insensitivity causes the classification under (AA).

The combinations (AS) and (SS) have been extensively studied in process theory; we refer to Austry & Boudol [2] and De Simone [28] for a comparison between MEIJE and SCCS, to Milner [23,24] and Hennessy [14] for CCS and SCCS, to Bergstra & Klop [7] for ACP and ASP, to De Bakker & Zucker [3,4] for uniform processes, and to Brookes [11,12], Winskel [30] for discussions about and comparisons between CSP and CCS. For CSP see Hoare [15,16] and Hoare e.a. [17].

The combination (AA) is studied for instance using temporal logic in Pnueli [26], Lamport [21] and Koymans, Vytopil & de Roever [19], Kuiper & de Roever [20]. Moreover, trace theories are used to describe the semantics of data flow networks (see Kahn [18], Brock & Ackerman [10]) and the semantics of restoring circuit logic (see Ebergen [13], Rem [27] and Van de Snepscheut [29]).

A discussion of the case (AA) in an algebraic setting is absent to our knowledge. In Milne [22] and Bergstra & Klop [6] the (AA) case is reduced to the (AS) case for switching resp. data flow. We are not aware of any "direct" algebraic descriptions of the (AA) case.

We will now proceed with the description of *three mechanisms for asynchronous communication (consistent with asynchronous co-operation).*
One may imagine a wild variety of different mechanisms for asynchronous communication. Three typical mechanisms which are strongly related to one another are
(i) Mail via an order preserving channel (queue)
(ii) Mail through a non-order-preserving channel (bag)
(iii) Causality ("action a *causes action* b*").*
For these cases we will present syntax in the form of a special purpose alphabet of atomic actions as well as an appropriate encapsulation operator which detrmines the semantics of the mechanism. In all three cases the semantics is given by an axiom system which extends the axiom system PA of Section 1. The first two cases will be the subject of Section 3, the third case is treated in Section 4.

3. MAIL VIA A CHANNEL

We will treat the cases of mail via an order-preserving channel and mail via a non-order-preserving channel together since the syntax and axioms proposed for these mechanisms coincide to a large extent.

3.1. The alphabet. Let B be a finite set of actions, D a finite set of data, c a special symbol (for 'channel'). For all $d \in D$ there are actions

$c \uparrow d$ *(send* d *via channel* c*: potential action)*

$c \Uparrow d$ *(send* d *via* c*: past action)*

$c \downarrow d$ *(receive* d *via* c*: potential action)*

$c \Downarrow d$ *(receive* d *via* c*: past action)*

The distinction between $c \uparrow d$ and $c \Uparrow d$ may be slightly unusual; $c \uparrow d$ indicates an intended (potential, future) action while $c \Uparrow d$ denotes a realised (actual, past) action. Likewise for $c \downarrow d$ and $c \Downarrow d$.

(Note that this distinction is implicitly also present in ACP: there a communication has the form $a|b = c$. Now the communication actions can be seen as potential actions, while the communication result c is an actual action.)

Notation: $c \uparrow D = \{c \uparrow d \mid d \in D\}$. Likewise for $c \Uparrow D$, etc.

The actions $b \in B$ are not related to channel c. (Although we specify syntax and axioms for one channel c only, the presence of several channels c,c',... is entirely unproblematic; in that case, B may also contain actions $c' \uparrow d$ etc. since also these are not related to channel c.)
Now we define the alphabet to be

$$A = B \cup \{\delta\} \cup (c\uparrow D) \cup (c\Uparrow D) \cup (c\downarrow D) \cup (c\Downarrow D) .$$

(Note that the cardinality $\#(A) = \#(B) + 4.\#(D) + 1.$)

3.2. Encapsulation. Here the situation diverges into the cases 'mail via an order-preserving channel' (3.2.1) and 'mail via an non-order-preserving channel' (3.2.2).
3.2.1. Let D* be the set of *sequences* σ of data $d \in D$. The empty sequence is denoted by ε. Concatenation of sequences σ,τ is denoted as $\sigma * \tau$; especially if $\sigma = \langle d_1,\ldots,d_n\rangle$ $(n \geq 0)$ then $d * \sigma = \langle d,d_1,\ldots,d_n\rangle$, and $\sigma * d = \langle d_1,\ldots,d_n,d\rangle$. Further, if $n \geq 1$, $\mathrm{last}(\sigma) = d_n$.

Now for each $\sigma \in D^*$ there is an *encapsulation operator* $\mu_c^\sigma: \mathcal{P} \to \mathcal{P}$ where $\mathcal{P}$ is the domain of processes (i.e. the elements of a process algebra satisfying the axioms below). If x is a process, then $\mu_c^\sigma(x)$ denotes the process obtained by requiring that the channel c is initially containing a data sequence σ and that no communications with c are performed outside x.
Phrased otherwise: $\mu_c^\sigma(x)$ is the result of *partial execution* of x w.r.t. c

with initial contents σ. (Here 'execution' refers to the transformation of a potential action like $c \uparrow d$ into a past action $c \Uparrow d$; moreover such a transformation has a side-effect on the contents of c as specified by the axioms below.) Another way of viewing the difference between $c \uparrow d$ and $c \Uparrow d$, etc. and between x and $\mu_c^\sigma(x)$ is in the distinction 'internal vs. external': the former $(c \uparrow d, x)$ is the internal view, the latter $(c \Uparrow d, \mu_c^\sigma(x))$ the external view. We will return to this matter in Remark 4.7 below.

3.2.2. For the bag-like channel the situation is very much the same except that a data sequence σ is now a *multiset* of data. We denote a finite multiset of $d \in D$ by 'M'. Now for all finite multisets M over D we introduce again an encapsulation or partial execution operator

$$\mu_c^M : \mathcal{P} \to \mathcal{P}$$

3.3. The signature. Although the various ingredients of the signature, both for the case of mail via a queue-like channel and via a bag-like channel, have now all been introduced, we will display these signatures once more in Table 2:

$+$	*alternative composition (sum)*
$\cdot$	*sequential composition (product)*
$\Vert$	*parallel composition (merge)*
$\mathbin{\Vert\!\!\lfloor}$	*left-merge*
δ	*dead-lock or failure*
b	*atomic action* $\in$ B, *independent from* c
$c \uparrow d$	*send* d *via channel* c; *internal view*
$c \Uparrow d$	*send* d *via channel* c; *external view*
$c \downarrow d$	*receive* d *via* c; *internal view*
$c \Downarrow d$	*receive* d *via* c; *external view*
μ_c^σ	*encapsulation w.r.t. queue-like channel* c
μ_c^M	*encapsulation w.r.t. bag-like channel* c

Table 2.

3.4. <u>Semantics</u>. Suppose a set B of actions, a set D of data and a channel name c are given. Then we have the following axiom systems:

$PA_\delta(\mu_c^\sigma, B, D)$ in Table 3 (next page)

$PA_\delta(\mu_c^M, B, D)$ in Table 4

for 'mail via a queue-like channel' and 'mail via a bag-like channel', respectively. Here 'a' varies over the alphabet $A = B \cup \{\delta\} \cup c{\uparrow}D \cup c{\Uparrow}D \cup c{\downarrow}D \cup c{\Downarrow}D$, and 'e' varies over $E = B \cup \{\delta\} \cup c{\Uparrow}D \cup c{\Downarrow}D$.

The axiom systems $PA_\delta(\mu_c^\sigma,B,D)$ and $PA_\delta(\mu_c^M,B,D)$ determine initial algebras

$A_\omega(+,\cdot,\|,\mathbin{|\!\lfloor},\delta,\mu_c^\sigma,B,D)$

$A_\omega(+,\cdot,\|,\mathbin{|\!\lfloor},\delta,\mu_c^M,B,D)$

respectively. These are just enrichments of the initial algebra

$A_\omega(+,\cdot,\|,\mathbin{|\!\lfloor},\delta)$

of PA_δ. Using a projective limit construction as in [7], or a topological completion as in [3,4], one finds larger models

$A^\infty(+,\cdot,\|,\mathbin{|\!\lfloor},\delta,\mu_c^\sigma,B,D)$

$A^\infty(+,\cdot,\|,\mathbin{|\!\lfloor},\delta,\mu_c^M,B,D)$

with infinite processes, in which all guarded systems of equations can be solved.

$PA_{\delta}(\mu_c^{\sigma}, B, D)$

$x + y = y + x$	A1
$(x + y) + z = x + (y + z)$	A2
$x + x = x$	A3
$(x + y)z = xz + yz$	A4
$(xy)z = x(yz)$	A5
$x + \delta = x$	A6
$\delta x = \delta$	A7
$x \| y = x \mathbin{\|\!\lfloor} y + y \mathbin{\|\!\lfloor} x$	M1
$a \mathbin{\|\!\lfloor} x = ax$	M2
$ax \mathbin{\|\!\lfloor} y = a(x \| y)$	M3
$(x + y) \mathbin{\|\!\lfloor} z = x \mathbin{\|\!\lfloor} z + y \mathbin{\|\!\lfloor} z$	M4
$\mu_c^{\sigma}(e) = e$	MO1
$\mu_c^{\sigma}(ex) = e.\mu_c^{\sigma}(x)$	MO2
$\mu_c^{\sigma}(c{\uparrow}d) = c{\Uparrow}d$	MO3
$\mu_c^{\sigma}(c{\uparrow}d \,.\, x) = c{\Uparrow}d \,.\, \mu_c^{d*\sigma}(x)$	MO4
$\mu_c^{\sigma*d}(c{\downarrow}d) = c{\Downarrow}d$	MO5
$\mu_c^{\sigma*d}(c{\downarrow}d \,.\, x) = c{\Downarrow}d \,.\, \mu_c^{\sigma}(x)$	MO6
$\mu_c^{\sigma}(c{\downarrow}d) = \delta$ if $d \neq \mathrm{last}(\sigma)$ or $\sigma = \varepsilon$	MO7
$\mu_c^{\sigma}(c{\downarrow}d \,.\, x) = \delta$ if $d \neq \mathrm{last}(\sigma)$ or $\sigma = \varepsilon$	MO8
$\mu_c^{\sigma}(x + y) = \mu_c^{\sigma}(x) + \mu_c^{\sigma}(y)$	MO9

Table 3. $(a \in A,\ e \in E,\ \sigma \in D^*)$

$PA_\delta(\mu_c^M, B, D)$

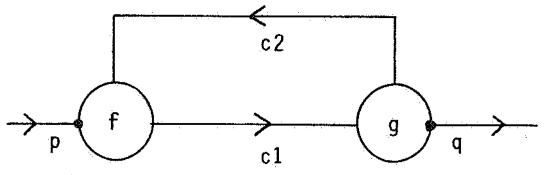
$x + y = y + x$	A1
$(x + y) + z = x + (y + z)$	A2
$x + x = x$	A3
$(x + y)z = xz + yz$	A4
$(xy)z = x(yz)$	A5
$x + \delta = x$	A6
$\delta x = \delta$	A7
$x \| y = x \mathbin{\|\!\lfloor} y + y \mathbin{\|\!\lfloor} x$	M1
$a \mathbin{\|\!\lfloor} x = ax$	M2
$ax \mathbin{\|\!\lfloor} y = a(x \| y)$	M3
$(x + y) \mathbin{\|\!\lfloor} z = x \mathbin{\|\!\lfloor} z + y \mathbin{\|\!\lfloor} z$	M4
$\mu_c^M(e) = e$	MN01
$\mu_c^M(ex) = e.\mu_c^M(x)$	MN02
$\mu_c^M(c\uparrow d) = c\Uparrow d$	MN03
$\mu_c^M(c\uparrow d\ .\ x) = c\Uparrow d\ .\ \mu_c^{M\cup\{d\}}(x)$	MN04
$\mu_c^{M\cup\{d\}}(c\downarrow d) = c\Downarrow d$	MN05
$\mu_c^{M\cup\{d\}}(c\downarrow d\ .\ x) = c\Downarrow d\ .\ \mu_c^M(x)$	MN06
$\mu_c^M(c\downarrow d) = \delta$ if $d \notin M$	MN07
$\mu_c^M(c\downarrow d\ .\ x) = \delta$ if $d \notin M$	MN08
$\mu_c^M(x + y) = \mu_c^M(x) + \mu_c^M(y)$	MN09

Table 4. ($a \in A$, $e \in E$, M a multiset over D)

3.5. Examples. We will now give some examples both for the case of an order-preserving channel and the case of a non-order-preserving channel.

Example (1), for a queue-like channel. Consider the following very simple data flow network:

Figure 1.



with actions

rp(d)	*(processor* f *reads value* d *at* p*)*
wq(d)	*(processor* g *writes* d *at port* q*)*

There are two order-preserving channels c1 and c2. Node f satisfies

$$f = \sum_{d \in D} (rp(d) + c2{\downarrow}d) \cdot c1{\uparrow}d \cdot f.$$

So, node f merges the inputs from p and c2 and emits these through c1. Node g is defined by

$$g = \sum_{d \in D} c1{\downarrow}d \cdot (i \cdot c2{\uparrow}\alpha(d) + i \cdot wq(d)) \cdot g$$

(The effect of the internal step i is to make the choice nondeterministic.) Here $\alpha: D \to D$ is a transformation of the data; g obtains d from c1 and then chooses whether to 'recycle' α(d) via c2 or to output d via port q.

The network N is now described by

$$N = \mu^{\varepsilon}_{c1}\, \mu^{\varepsilon}_{c2}\, (f \parallel g).$$

Note that the actions $c1{\downarrow}d$, $c1{\uparrow}d$, $c1{\Downarrow}d$ and $c1{\Uparrow}d$ are unrelated to c2 and thereby work as b's in the definition for μ^{σ}_{c2}. Conversely, the send and receive actions for c2 are unrelated to c1.

Example (2), for a queue-like channel. Consider the very simple communication protocol as in Figure 2:

T:

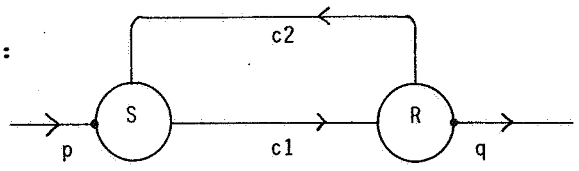


Figure 2.

$$S = \sum_{d \in D} rp(d) \cdot c1{\uparrow}d \cdot c2{\downarrow}ack \cdot S$$

$$R = \sum_{d \in D} c1{\downarrow}d \cdot wq(d) \cdot c2{\uparrow}ack \cdot R$$

$$T = \mu^{\varepsilon}_{c1}\ \mu^{\varepsilon}_{c2}\ (S \| R).$$

In fact the entire protocol T satisfies the following recursion equation (as one easily computes from the axioms in $PA_{\delta}(\mu^{\sigma}_{c}, B, D)$):

$$T = \sum_{d \in D} rp(d) \cdot c1{\Uparrow}d \cdot c1{\Downarrow}d \cdot wq(d) \cdot c2{\Uparrow}ack \cdot c2{\Downarrow}ack \cdot T.$$

<u>Example (3)</u>, for a bag-like channel:

(i) $\mu^{\emptyset}_{c}(c{\uparrow}d \cdot c{\downarrow}d) = c{\Uparrow}d \cdot c{\Downarrow}d$

(ii) $\mu^{\emptyset}_{c}(c{\uparrow}d \cdot \sum_{u \in D} c{\downarrow}u) = c{\Uparrow}d \cdot c{\Downarrow}d$

(iii) $\mu^{\emptyset}_{c}(c{\uparrow}d \ \| \ c{\downarrow}d) = c{\Uparrow}d \cdot c{\Downarrow}d$

(iv) $\mu^{\emptyset}_{c}(c{\uparrow}d1 \cdot c{\uparrow}d2 \cdot \sum_{u \in D} c{\downarrow}u \cdot \sum_{u \in D} c{\downarrow}u) =$

$c{\Uparrow}d1 \cdot c{\Uparrow}d2 \cdot (c{\Downarrow}d1 \cdot c{\Downarrow}d2 + c{\Downarrow}d2 \cdot c{\Downarrow}d1)$

(v) Let $D = D1 \cup D2$, $D1 \cap D2 = \emptyset$,

and $H = \left\{ \sum_{d \in D1} c1{\downarrow}d \cdot c2{\uparrow}d + \sum_{d \in D2} c1{\downarrow}d \cdot c3{\uparrow}d \right\} \cdot H$

Then H separates the D1 messages from the D2 messages.

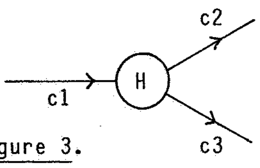


Figure 3.

(vi) Let d1 $\neq$ d2. Then:

$$\mu_c^{\emptyset}(c{\uparrow}d1 \cdot c{\downarrow}d2) = c{\Uparrow}d1 \cdot \delta$$

$$\mu_c^{\emptyset}(c{\uparrow}d1 \parallel c{\downarrow}d2) = c{\Uparrow}d1 \cdot \delta$$

$$\mu_c^{\emptyset}(c{\downarrow}d2 \mathbin{\lfloor\!\lfloor} c{\uparrow}d1) = \delta.$$

3.6. Remark. Notice that there is no guarantee that after a send action c↑d the corresponding receive action c↓d will ever be performed. Thus the send action *enables* the receive action but does not *force* its execution. This holds for both mechanisms: queue-like channel and bag-like channel.

3.7. Remark. In the tele-communications area the design language SDL, used by CCITT, is quite popular. SDL mainly consists of a format for graphical notations for concurrent system descriptions with a send and receive mechanism. SDL leaves open the nature of the transmission protocol that supports the send and receive instructions. In SDL, example 3.5(2) can be depicted as follows:

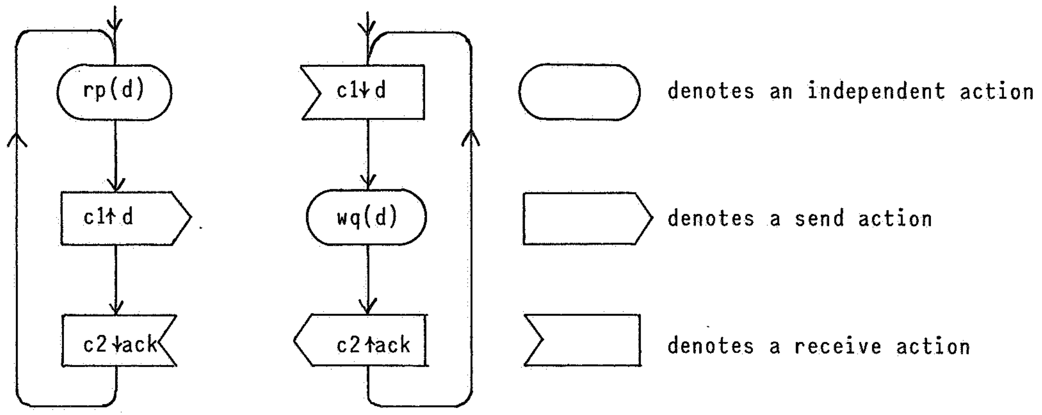


Figure 4.

Here it is assumed that in each cycle d receives a value at rp(d) and c1↓d respectively. μ-Encapsulation of the protocol leads to the following SDL description:

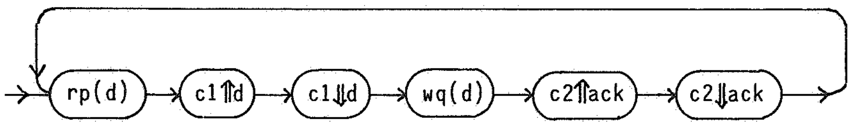


Figure 5.

3.8. Remark on synchronous communication.

A syntax for synchronous communication along a channel c, inspired by CSP and CCS, would be:

c!d	*send* d
c?d	*receive* d
c#d	*communicate* d

In ACP [7] one introduces a communication function | on actions. In this particular example, | would work as follows:

$$c!d \mid c?d = c\#d.$$

Note: (i) We do not use variables; i.e. $c?x \cdot P$ is modeled by $\sum_{d \in D} c?d \cdot P[d/x]$.
(ii) This differs from CCS where one would have

$$c(d) \mid \bar{c}(d) = \tau.$$

4. CAUSALITY

In the previous section, the action $c\Uparrow d$ is the 'actualised form' of $c\uparrow d$ and likewise $c\Downarrow d$ is $c\downarrow d$ after execution or actualisation. Moreover, a causal effect is involved: $c\uparrow d$ causes $c\downarrow d$. These concepts will be made explicit in the present section.

4.1. Actualisation.

On the alphabet A we postulate an operator $\hat{}: A \to A$, such that $\hat{\delta} = \delta$ and $\hat{\hat{a}} = \hat{a}$ for all $a \in A$. The action $\hat{a}$ is called the *actualisation* of a. Writing $B = A - \hat{A}$, where $\hat{A} = \{\hat{a} \mid a \in A\}$, A is partitioned as follows:

$$A = B \cup \hat{A}.$$

4.2. Causal relations.

On the set B of not yet completed actions we have a binary relation R encoding the causal relations between such actions. Instead of $(a,b) \in R$ we write:

$$a \Vdash b,$$

in words: "a causes b". Further notations are:
Dom(R) for the *domain* of R, i.e. $\mathrm{Dom}(R) = \{b \mid \exists b' \; b' \Vdash b\}$, and
Ran(R) for the *range* of R, i.e. $\mathrm{Ran}(R) = \{b \mid \exists b' \; b' \Vdash b\}$.
So Dom(R) contains the 'causes' or 'stimuli' and Ran(R) the 'effects' or 'res-

ponses'. Note that an action can be both a cause and an effect. Finally, $R(b) = \{b' \mid b \,|\!\vdash b'\}$, the set of effects of b.

4.3. <u>Encapsulation</u>. Let $b \in B$. Performing b has two consequences: b is changed into $\hat{b}$, and all $b' \in R(b)$ (i.e. caused by b) are now *enabled*. The operator which takes care of the execution of b (or in another phrasing, which changes the view from 'internal' to 'external') and which takes into account which actions are enabled, is the *encapsulation operator* γ^E. Here $E \subseteq B$. The intuitive meaning of γ^E is: $\gamma^E(x)$ is the process where all causal effects take place within x, i.e. actions within x are neither enabled nor disabled by actions outside x and conversely. Moreover, initially the actions $\in E$ are enabled.

4.4. <u>Semantics</u>. We will now give axioms for the operations γ^E. On the initial algebra $I(PA_\delta)$ these equations, in Table 5 below, specify an enrichment with operators γ^E; in the general case the equations are to be seen as additional axioms. As with the previous axiomatisations for the more important model constructions it is clear how to enrich these with an interpretation of the operators γ^E.

$PA_\delta(\gamma,\hat{\ })$ over atoms A with causality relation R

$x + y = y + x$	A1
$(x + y) + z = x + (y + z)$	A2
$x + x = x$	A3
$(x + y)z = xz + yz$	A4
$(xy)z = x(yz)$	A5
$x + \delta = x$	A6
$\delta x = \delta$	A7
$x \| y = x \lfloor\!\lfloor y + y \lfloor\!\lfloor x$	M1
$a \lfloor\!\lfloor y = ay$	M2
$ax \lfloor\!\lfloor y = a(x \| y)$	M3
$(x + y) \lfloor\!\lfloor z = x \lfloor\!\lfloor z + y \lfloor\!\lfloor z$	M4
$\hat{\delta} = \delta$	G1
$\hat{\hat{a}} = \hat{a}$	G2
$\gamma^E(a) = \hat{a}$ if $a \in E$ or $a \notin Ran(R)$	G3
$\gamma^E(a) = \delta$ if $a \notin E$ and $a \in Ran(R)$	G4
$\gamma^E(ax) = \gamma^E(a) \cdot \gamma^{(E - \{a\}) \cup R(a)}(x)$	G5
$\gamma^E(x + y) = \gamma^E(x) + \gamma^E(y)$	G6

<u>Table 5.</u>

4.5. Example. (i) Suppose $a|\!\vdash d$, $c|\!\vdash b$ (see Figure 6(a)). Then $\gamma^{\emptyset}(ab\|cd) = \gamma^{\emptyset}(a(b\|cd)) + \gamma^{\emptyset}(c(d\|ab)) = \hat{a}\gamma^{\{d\}}(b\|cd) + \ldots =$ $\hat{a}\gamma^{\{d\}}(bcd + c(d\|b)) + \ldots = \hat{a}(\delta + \hat{c}\gamma^{\{d,b\}}(db + bd)) + \ldots =$ $\hat{a}\hat{c}(\hat{d}\hat{b} + \hat{b}\hat{d}) + \hat{c}\hat{a}(\hat{b}\hat{d} + \hat{d}\hat{b}) = (\hat{a}\|\hat{c})(\hat{b}\|\hat{d})$.

(ii) Suppose $d|\!\vdash a$, $b|\!\vdash c$ (see Figure 6(b)). Then $\gamma^{\emptyset}(ab\|cd) = \delta$.

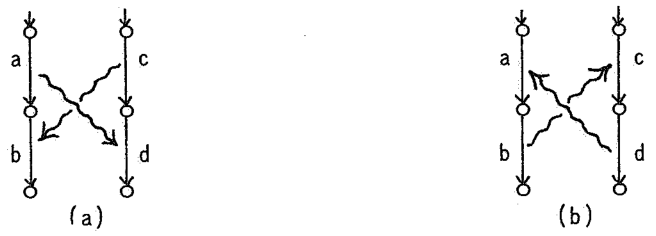


Figure 6.

Note that circular causal relations (such as in this example (ii)) yield deadlock. Here an action a must be considered to cause also the actions accessible from a ('later' than a). (Indeed, we have $a{\bullet}b = \gamma^{\emptyset}(a\|b)$ for $a|\!\vdash b$.)

(iii) Let X and Y be the two infinite processes recursively defined by $X = abX$ and $Y = cdY$; so $X = (ab)^{\omega}$ and $Y = (cd)^{\omega}$. Suppose $a|\!\vdash c$ and $d|\!\vdash b$. Then

$$\gamma^{\emptyset}(X\|Y) = \gamma^{\emptyset}(a(bX\|Y) + c(dY\|X)) = \hat{a}\gamma^{\{c\}}(bX\|Y) + \delta =$$
$$\hat{a}\gamma^{\{c\}}(b(X\|Y) + c(dY\|bX)) = \hat{a}(\delta + \hat{c}\gamma^{\emptyset}(dY\|bX)) =$$
$$\hat{a}\hat{c}(\hat{d}\gamma^{\{b\}}(Y\|bX) + \delta) = \hat{a}\hat{c}\hat{d}\gamma^{\{b\}}(b(X\|Y) + Y \lfloor\!\lfloor bX) =$$
$$\hat{a}\hat{c}\hat{d}\hat{b}\gamma^{\emptyset}(X\|Y).$$

Hence $\gamma^{\emptyset}(X\|Y) = (\hat{a}\hat{c}\hat{d}\hat{b})^{\omega}$.

4.6. Remark. (i) It should be noted that however often an action b has been enabled, after being performed it is again disabled. For instance if $b|\!\vdash c$, then

$$\gamma^{\emptyset}(bbcc) = \hat{b}\gamma^{\{c\}}(bcc) = \hat{b}\hat{b}\gamma^{\{c\}}(cc) = \hat{b}\hat{b}\hat{c}\gamma^{\emptyset}(c) = \hat{b}\hat{b}\hat{c}\delta.$$

Due to this interpretation of causality as introducing an obligation (which has no multiplicity), the mail via an unordered channel mechanism differs from the present mechanism. For, in the setting of Section 3 we have

$$\mu_c^{\emptyset}(c{\uparrow}d \cdot c{\uparrow}d \cdot c{\downarrow}d \cdot c{\downarrow}d) = c{\Uparrow}d \cdot c{\Uparrow}d \cdot c{\Downarrow}d \cdot c{\Downarrow}d.$$

It is, however, easy to specify the variant of the causality mechanism above such that the obligations form a multiset rather than a set: axioms G1-6 from Table 5 carry over to that case unaltered, with as only stipulation that E is now a multiset.
(ii) It is also simple to generalize the above causality relation to the case where an effect b may have several causes $a_1,\ldots,a_n$:

$$a_1,\ldots,a_n \Vdash b,$$

meaning that all the a_i $(i=1,\ldots,n)$ have to be executed in order to enable b. We will not do so, here.

4.7. Remark. There is an interesting connection between the 'spatial' notion of encapsulation (as effectuated by the operators ∂_H in ACP, μ_c^{σ}, μ_c^{M} in the mail mechanisms of Section 3 and the present γ^E for causality) and the 'temporal' notion of execution. In some sense, one could say:

encapsulation = execution.

Indeed, an encapsulated process can be thought to be already executed since no further interactions with an environment are possible.

4.8. Example. We will, as a conclusion of this paper, now discuss a somewhat involved example. This example constitutes an abstract version of the highest level of a case study specification as reported in [5]. Henk Obbink [25] from Philips Research suggested us to use a stimulus-response (or causality) mechanism at the highest specification level. The motivation for the present paper is just to present a proper foundation in process algebra for the causality mechanism. In fact, mail via order-preserving or non-order-preserving channels turned out to be minor modifications on the same theme (with the advantage of having better syntax).

Let us consider a configuration of three components:

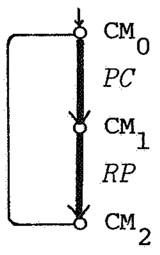
CM	*command module*
P	*printer*
D	*display*

The only command that CM can issue is to start the printer; the printer will stop by itself. If the printer runs out of paper, a message to this effect must be displayed whereafter new paper will be provided, and printing proceeds. When printing has finished this is reported to CM.

The behaviour of the components is depicted in the diagrams in Figure 7(a),(b),(c). From now on, we adopt the following

Convention. *We will use the following typographical convention: instead of denoting actions as* b, $\hat{b}$ *we will write, respectively, b and* b. *So italicized actions are* $\in B$ *(not yet completed) and* completed actions $\in \hat{B}$ *are in usual print.*

CM:

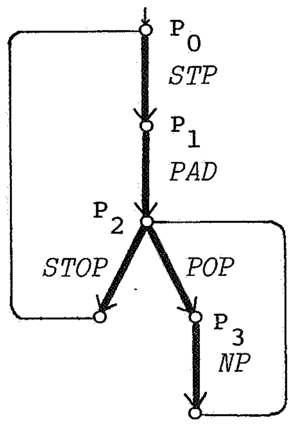


(print command)

(reply from printer)

Figure 7(a).

P:



(start printer)

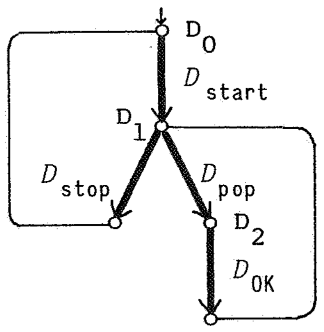
(printer asks data)

(printer stops) (printer runs out of paper)

(printer receives new paper)

Figure 7(b).

D:



(display "start")

(display "stop") (display "pop")

(display "OK")

Figure 7(c).

In these diagrams, fat arrows represent actions; the other lines identify control points and have no direction.

The causal relations $\in R$ are listed below:

$$\begin{array}{llll} PC \Vdash STP & & STP \Vdash D_{start} \\ STOP \Vdash D_{stop} & & D_{stop} \Vdash RP \\ POP \Vdash D_{pop} & & D_{pop} \Vdash NP \\ NP \Vdash D_{OK} & & \end{array}$$

The entire system $\mathcal{S}$ is now described by:

$$\mathcal{S} = \gamma^{\emptyset}(CM_0 \| P_0 \| D_0).$$

Further, let

$$\mathcal{S}^* = \gamma^{\{NP\}}(CM_1 \| P_3 \| D_2),$$

then it can easily be shown that $\mathcal{S}$ and $\mathcal{S}^*$ satisfy the following recursion equations:

$$\begin{cases} \mathcal{S} = PC \cdot STP \cdot [D_{start} \cdot PAD \cdot \{STOP \cdot D_{stop} \cdot RP \cdot \mathcal{S} + POP \cdot D_{pop} \cdot \mathcal{S}^*\} + \\ \qquad + PAD \cdot \{D_{start} \cdot (STOP \cdot D_{stop} \cdot RP \cdot \mathcal{S} + POP \cdot D_{pop} \cdot \mathcal{S}^*) + \\ \qquad\qquad + STOP \cdot D_{stop} \cdot RP \cdot \mathcal{S} + POP \cdot D_{pop} \cdot \mathcal{S}^*\}] \\ \\ \mathcal{S}^* = NP \cdot [STOP \cdot D_{OK} \cdot D_{stop} \cdot RP \cdot \mathcal{S} + POP \cdot D_{OK} \cdot D_{pop} \cdot \mathcal{S}^* + \\ \qquad + D_{OK} \cdot (STOP \cdot D_{stop} \cdot RP \cdot \mathcal{S} + POP \cdot D_{pop} \cdot \mathcal{S}^*)] \end{cases}$$

REFERENCES

[1] ADJ (GOGUEN, J.A., THATCHER, J.W., WAGNER, E.G. & J.B. WRIGHT), *Initial algebra semantics and continuous algebras*, JACM Vol.24, Nr.1, p.68-95 (1975).

[2] AUSTRY, D. & G. BOUDOL, *Algèbre de processus et synchronisation*, Theoretical Computer Science 30 (1984), p.91-131.

[3] DE BAKKER, J.W. & J.I. ZUCKER, *Denotational semantics of concurrency*, Proc. 14th ACM Symp. on Theory of Computing, p.153-158, 1982.

[4] DE BAKKER, J.A. & J.I. ZUCKER,
Processes and the denotational semantics of concurrency,
Information and Control, Vol.54, No.1/2, p.70-120, 1982.

[5] BERGSTRA, J.A., HEERING, J., KLINT, P. & J.W. KLOP,
Een analyse van de case-study HμP,
mimeographed notes, Centrum voor Wiskunde en Informatica, Amsterdam 1984.

[6] BERGSTRA, J.A. & J.W. KLOP,
A process algebra for the operational semantics of static data flow networks,
Report IW 222/83, Mathematisch Centrum, Amsterdam 1983.

[7] BERGSTRA, J.A. & J.W. KLOP,
Process algebra for communication and mutual exclusion. Revised version,
Report CS-R8409, Centrum voor Wiskunde en Informatica, Amsterdam 1984.

[8] BERGSTRA, J.A., KLOP, J.W. & J.V. TUCKER,
Algebraic tools for system construction,
in: Logics of Programs, Proceedings 1983 (eds. E. Clarke and D. Kozen), Springer LNCS 164, 1984.

[9] BRANQUART, P., LOUIS, G. & P. WODON,
An analytical description of CHILL, the CCITT High Level Language,
Springer LNCS 128, 1982.

[10] BROCK, J.D. & W.B. ACKERMAN,
Scenarios: A model of non-determinate computation,
in: Proc. Formalization of Programming Concepts (eds. J. Díaz and I. Ramos), p.252-259, Springer LNCS 107, 1981.

[11] BROOKES, S.D.,
On the relationship of CCS and CSP,
Proc. 10th ICALP, Barcelona 1983 (ed. J. Díaz), Springer LNCS 154, p.83-96, 1983.

[12] BROOKES, S.D. & W.C. ROUNDS,
Behavioural equivalence relations induced by programming logics,
Proc. 10th ICALP, Barcelona 1983, Springer LNCS 154 (ed. J. Díaz), p.97-108, 1983.

[13] EBERGEN, J.,
On VLSI design,
NGI-SION Proceedings 1984, p.144-150, Nederlands Genootschap voor Informatica, Amsterdam 1984.

[14] HENNESSY, M.,
A term model for synchronous processes,
Information and Control 51, p.58-75 (1981).

[15] HOARE, C.A.R.,
Communicating Sequential Processes,
C.ACM 21 (1978), p.666-677.

[16] HOARE, C.A.R.,
A model for Communicating Sequential Processes,

in: "On the construction of programs" (eds. R.M. McKeag and A.M. McNaghton), Cambridge University Press (1980), p.229-243.

[17] HOARE, C., BROOKES, S. & W. ROSCOE,
A theory of communicating sequential processes,
Programming Research Group, Oxford University (1981). To appear in JACM.

[18] KAHN, G.,
The semantics of a simple language for parallel programming,
in: Proc. IFIP 74, North-Holland, Amsterdam 1974.

[19] KOYMANS, R., VYTOPIL, J. & W.P. DE ROEVER,
Real-time programming and asynchronous message passing,
in: Proc. of the Second Annual ACM Symposium on Principles of Distributed Computing, Montreal, 1983.

[20] KUIPER, R. & W.P. DE ROEVER,
Fairness assumptions for CSP in a temporal logic framework,
TC2 Working Conference on the Formal Description of Programming Concepts, Proc., Garmisch 1982.

[21] LAMPORT, L.
'Sometime' is sometimes 'NOT never',
tutorial on the temporal logics of programs, SRI International CSL-86, 1979.

[22] MILNE, G.J.,
CIRCAL: A calculus for circuit description,
Integration, Vol.1, No.2 & 3, 1983, p.121-160.

[23] MILNER, R.,
A Calculus of Communicating Systems,
Springer LNCS 92, 1980.

[24] MILNER, R.,
Calculi for synchrony and asynchrony,
Theor. Comp. Sci. 25 (1983), p.267-310.

[25] OBBINK, H., personal communication, April 1984.

[26] PNUELI, A.,
The temporal logic of programs,
in: Proc. 19th Ann. Symp. on Foundations of Computer Science, IEEE, p.46-57, 1977.

[27] REM, M.,
Partially ordered computations, with applications to VLSI design,
Proc. 4th Advanced Course on Foundations of Computer Science, Part 2 (eds. J.W. de Bakker and J. van Leeuwen), Mathematical Centre Tracts 159, p.1-44, Mathematisch Centrum, Amsterdam 1983.

[28] DE SIMONE, R.,
On MEIJE and SCCS: infinite sum operators vs. non-guarded definitions,
Theoretical Computer Science 30 (1984), p.133-138.

[29] VAN DE SNEPSCHEUT, J.L.A.,
Trace Theory and VLSI Design,
Ph.-D. Thesis, Eindhoven University of Technology, 1983.

[30] WINSKEL, G.,
Event structure semantics for CCS and related languages,
Proc. ICALP 82, Springer LNCS 140, p.561-576, 1982.

[31] WINSKEL, G.,
Synchronisation trees,
Proc. 10th ICALP (ed. J. Díaz), Barcelona 1983, Springer LNCS 154, p.695-711.