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
The Centre for Mathematics and Computer Science is a research institute of the Stichting Mathematisch Centrum, which was founded on February 11, 1946, as a nonprofit institution aiming at the promotion of mathematics, computer science, and their applications. It is sponsored by the Dutch Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O.).
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.

KEY WORDS & PHRASES: concurrency, communicating processes, asynchronous communication, stimulus-response mechanism.
NOTE: This report will be submitted for publication elsewhere.
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: PA₀

As a point of departure of this exposition let us consider the following axiom system PA₀ (see Table 1 below). A is a finite set of atomic actions, δ ∈ A represents deadlock or failure. There are four operators on processes:

- + alternative composition (sum)
- * sequential composition (product)
- || parallel composition (merge)
- L left-merge

The functionality of each of these binary operators is P × P → P, where P is the class of processes. All atomic actions are processes themselves.

The axiom system PA₀ (see [7] for PA; the axioms for δ are also in [7]) is this:

$$
\text{Table 1:}
\begin{array}{ll}
x + y = y + x & A1 \\
(x + y) + z = x + (y + z) & A2 \\
x + x = x & A3 \\
x(y + z) = xy + xz & A4 \\
(xy)z = x(yz) & A5 \\
x + \delta = x & A6 \\
\delta x = \delta & A7 \\
x||y = x||y + y||x & M1 \\
\alpha||x = \alpha x & M2 \\
(ax)||y = a(x||y) & M3 \\
(x + y)||z = x||z + y||z & M4
\end{array}
$$
Here $x, y, z$ range over $P$ and 'a' ranges over $A$. Products $x \cdot y$ are frequently written as $xy$. Let $\Sigma_{PA_\delta}$ be the signature of $PA_\delta$:

\[
\begin{array}{c|ccc}
\Sigma_{PA_\delta} & : & P & (sorts) \\
\text{IF:} & + : & P \times P + P & (functions) \\
& \cdot : & P \times P + P \\
& \| : & P \times P + P \\
& \|_l : & P \times P + P \\
\text{\&:} & a \in P & (for \ all \ a \in A) & (constants)
\end{array}
\]

There are at least two ways to consider $PA_\delta$:

(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

\[\text{Alg}(\Sigma_{PA_\delta}, PA_\delta).\]

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

**Definition.** (i) If $a = a_1 \ast a_2 \ast \ldots \ast a_n$, where $a_i \in A$ ($i = 1, \ldots, n$) and $\ast$ denotes concatenation, then $a$ 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

\[
P = p_1, \quad p_i = a_i p_{i+1} + q_i \quad (i = 1, \ldots, n-1)
\]

\[
p_n = a_n + q_n
\]

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

\[
P = p_1, \quad p_i = a_i p_{i+1} + q_i \quad (i \geq 1).
\]
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 $PA_\delta$ 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 $PA_\delta$ 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 actions 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 + asynchronous 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 determines 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 ∈ D there are actions
\[c \downarrow d\] (send \(d\) via channel \(c\): past action)

\[c \downarrow d\] (receive \(d\) via \(c\): past action)

The distinction between \(c \downarrow d\) and \(c \downarrow d\) may be slightly unusual; \(c \downarrow d\) indicates an intended (potential, future) action while \(c \downarrow d\) 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:** \(c \uparrow D = \{c \downarrow d \mid d \in D\}\). 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 \(c' \downarrow d\) etc. since also these are not related to channel \(c\).)

Now we define the alphabet to be

\[A = B \cup \{\delta\} \cup (c \downarrow D) \cup (c \uparrow D) \cup (c \downarrow D) \cup (c \uparrow D)\]

(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 \(\sigma\) of data \(d \in D\). The empty sequence is denoted by \(\varepsilon\). Concatenation of sequences \(\sigma, \tau\) is denoted as \(\sigma \ast \tau\); especially if \(\sigma = <d_1, \ldots, d_n>\) (\(n > 0\)) then \(d \ast \sigma = <d, d_1, \ldots, d_n>\), and \(\sigma \ast d = <d_1, \ldots, d_n, d>\). Further, if \(n \geq 1\), \(\text{last}(\sigma) = d_n\).

Now for each \(\sigma \in D^*\) there is an encapsulation operator \(\mu^\sigma_C: \mathcal{P} \rightarrow \mathcal{P}\) 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 \(\mu^\sigma_C(x)\) denotes the process obtained by requiring that the channel \(c\) is initially containing a data sequence \(\sigma\) and that no communications with \(c\) are performed outside \(x\).

Phrased otherwise: \(\mu^\sigma_C(x)\) is the result of partial execution of \(x\) w.r.t. \(c\).
with initial contents \( c \). (Here 'execution' refers to the transformation of a potential action like \( c + d \) 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 \( c + d \) and \( c \uparrow d \), etc. and between \( x \) and \( \mu^g_c(x) \) is in the distinction 'internal vs. external': the former \( (c + d, x) \) is the internal view, the latter \( (c \uparrow d, \mu^g_c(x)) \) 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 \( a \) is now a multiiset of data. We denote a finite multiiset of \( d \in D \) by 'M'. Now for all finite multisets \( M \) over \( D \) we introduce again an encapsulation or partial execution operator

\[
\mu^M_c : \rho \to \rho
\]

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:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>alternative composition (sum)</td>
</tr>
<tr>
<td>*</td>
<td>sequential composition (product)</td>
</tr>
<tr>
<td>|</td>
<td>parallel composition (merge)</td>
</tr>
<tr>
<td>[</td>
<td>left-merge</td>
</tr>
<tr>
<td>&amp;</td>
<td>dead-lock or failure</td>
</tr>
<tr>
<td>b</td>
<td>atomic action ( \in B ), independent from ( c )</td>
</tr>
<tr>
<td>( c + d )</td>
<td>send ( d ) via channel ( c ); internal view</td>
</tr>
<tr>
<td>( c \uparrow d )</td>
<td>send ( d ) via channel ( c ); external view</td>
</tr>
<tr>
<td>( c + d )</td>
<td>receive ( d ) via ( c ); internal view</td>
</tr>
<tr>
<td>( c \downarrow d )</td>
<td>receive ( d ) via ( c ); external view</td>
</tr>
<tr>
<td>( \mu^g_c )</td>
<td>encapsulation w.r.t. queue-like channel ( c )</td>
</tr>
<tr>
<td>( \mu^M_c )</td>
<td>encapsulation w.r.t. bag-like channel ( c )</td>
</tr>
</tbody>
</table>

Table 2.
3.4. **Semantics.** 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:

$$\text{PA}_\delta^\sigma(C, B, D)$$ in Table 3 (next page)

$$\text{PA}_\delta^M(C, B, D)$$ in Table 4

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

The axiom systems $\text{PA}_\delta^\sigma(C, B, D)$ and $\text{PA}_\delta^M(C, B, D)$ determine initial algebras

$$A_\omega(\cdot, \cdot, \parallel, \delta, \mu_C^\sigma, B, D)$$

$$A_\omega(\cdot, \cdot, \parallel, \delta, \mu_C^M, B, D)$$

respectively. These are just enrichments of the initial algebra

$$A_\omega(\cdot, \cdot, \parallel, \delta)$$

de $\text{PA}_\delta$. Using a projective limit construction as in [7], or a topological completion as in [3,4], one finds larger models

$$A_\omega^\omega(\cdot, \cdot, \parallel, \delta, \mu_C^\sigma, B, D)$$

$$A_\omega^\omega(\cdot, \cdot, \parallel, \delta, \mu_C^M, B, D)$$

with infinite processes, in which all guarded systems of equations can be solved.
Table 3. \((a \in A, e \in E, \sigma \in D^*)\)
Table 4. (a ∈ A, e ∈ 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.](image)

with actions

- \( \text{rp}(d) \) (processor \( f \) reads value \( d \) at \( p \))
- \( \text{wq}(d) \) (processor \( g \) writes \( d \) at port \( q \))

There are two order-preserving channels \( c_1 \) and \( c_2 \). Node \( f \) satisfies

\[
f = \sum_{d \in D} (\text{rp}(d) + c_2 + d) \cdot c_1 + d \cdot f.
\]

So, node \( f \) merges the inputs from \( p \) and \( c_2 \) and emits these through \( c_1 \).

Node \( g \) is defined by

\[
g = \sum_{d \in D} c_1 + d \cdot (i \cdot c_2 + a(d) + i \cdot \text{wq}(d)) \cdot g
\]

(The effect of the internal step \( i \) is to make the choice nondeterministic.)

Here \( a : D + D \) is a transformation of the data; \( g \) obtains \( d \) from \( c_1 \) and then chooses whether to 'recycle' \( a(d) \) via \( c_2 \) or to output \( d \) via port \( q \).

The network \( N \) is now described by

\[
N = \mu_{c_1}^\varepsilon \mu_{c_2}^\varepsilon (f \parallel g).
\]

Note that the actions \( c_1 + d, c_1 + d, c_1 + d \) and \( c_1 + d \) are unrelated to \( c_2 \) and thereby work as \( b \)'s in the definition for \( \mu_{c_2}^\sigma \). Conversely, the send and receive actions for \( c_2 \) are unrelated to \( c_1 \).

**Example (2),** for a queue-like channel. Consider the very simple communication protocol as in Figure 2:
In fact the entire protocol $T$ satisfies the following recursion equation (as one easily computes from the axioms in $PA_0(\mu_C,B,D)$):

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

Example (3), for a bag-like channel:

(i) $\mu_C(c+d \cdot c+d) = c+d \cdot c+d$

(ii) $\mu_C(c+d \cdot \sum_{u \in D} c+u) = c+d \cdot c+d$

(iii) $\mu_C(c+d \mid c+d) = c+d \cdot c+d$

(iv) $\mu_C(c+d_1 \cdot c+d_2 \cdot \sum_{u \in D} c+u \cdot \sum_{u \in D} c+u) = c+d_1 \cdot c+d_2 \cdot (c+d_2 \cdot c+d_2 + c+d_2 \cdot c+d_2)$

(v) Let $D = D_1 \cup D_2$, $D_1 \cap D_2 = \emptyset$,

and $H = \left\{ \sum_{d \in D_1} c1+d \cdot c2+d + \sum_{d \in D_2} c1+d \cdot c3+d \right\} \cdot H$

Then $H$ separates the $D_1$ messages from the $D_2$ messages.
Let $d_1 \neq d_2$. Then:

\[
\begin{align*}
\mu^0_c(c+d_1 \cdot c+d_2) &= c\diamond d_1 \cdot \delta \\
\mu^0_c(c+d_1 \mid\mid c+d_2) &= c\diamond d_1 \cdot \delta \\
\mu^0_c(c+d_2 \mid\mid c+d_1) &= \delta.
\end{align*}
\]

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:

![Diagram 4](image)

**Figure 4.**

Here it is assumed that in each cycle $d$ receives a value at $rp(d)$ and $c+d$ respectively. $\mu$-Encapsulation of the protocol leads to the following SDL description:

![Diagram 5](image)

**Figure 5.**
3.8. Remark on synchronous communication.

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

\[ \begin{align*}
  c!d & \quad \text{send } d \\
  c?d & \quad \text{receive } d \\
  c\#d & \quad \text{communicate } d
\end{align*} \]

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. \(c?x.P\) is modeled by \(\sum_{d \in D} c?d.P[d/x]\).

(ii) This differs from CCS where one would have

\[ c(d) \mid \overline{c(d)} = \tau. \]

4. CAUSALITY

In the previous section, the action \(c\#d\) is the 'actualised form' of \(c!d\) and likewise \(c\&d\) is \(c!d\) after execution or actualisation. Moreover, a causal effect is involved: \(c!d\) causes \(c\&d\). These concepts will be made explicit in the present section.

4.1. Actualisation. On the alphabet \(A\) we postulate an operator \(\hat{\cdot}\): \(A \rightarrow A\), such that \(\emptyset = \hat{\emptyset}\) and \(\hat{a} = \hat{a}\) 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:

\[ a \models\ b, \]

in words: "a causes b". Further notations are:

- \(\text{Dom}(R)\) for the domain of \(R\), i.e. \(\text{Dom}(R) = \{b \mid \exists b' \ b' \models \ b\}\), and
- \(\text{Ran}(R)\) for the range of \(R\), i.e. \(\text{Ran}(R) = \{b \mid \exists b' \ b' \models \ b\}\).

So \(\text{Dom}(R)\) contains the 'causes' or 'stimuli' and \(\text{Ran}(R)\) the 'effects' or 'res-
pones'. 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. Encapsulation. Let \( b \in B \). Performing \( b \) has two consequences: \( b \) is changed into \( b' \), 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 \( \gamma^E \). Here \( E \subseteq B \). The intuitive meaning of \( \gamma^E \) 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 \( a \in E \) are enabled.

4.4. Semantics. We will now give axioms for the operations \( \gamma^E \). On the initial algebra \( I(PA_0) \) these equations, in Table 5 below, specify an enrichment with operators \( \gamma^E \); 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 \( \gamma^E \).

\[
\begin{align*}
\text{Table 5.} \\
\begin{array}{ll}
\text{PA}_0(\gamma^E, \cdot) \text{ over atoms } A \text{ with causality relation } R \\

\begin{array}{ll}
x + y = y + x & \text{A1} \\
(x + y) + z = x + (y + z) & \text{A2} \\
x + x = x & \text{A3} \\
(x + y)z = xz + yz & \text{A4} \\
(xy)z = x(yz) & \text{A5} \\
x + \delta = x & \text{A6} \\
\delta x = \delta & \text{A7} \\
x \| y = x + y & \text{M1} \\
\text{M2} \\
a \| y = ay & \text{M2} \\
ax \| y = a(x\|y) & \text{M3} \\
(x + y) \| z = x \| z + y \| z & \text{M4} \\
\text{} \\
\delta = \delta & \text{G1} \\
\hat{a} = \hat{a} & \text{G2} \\
\gamma^E(a) = \delta \text{ if } a \in E \text{ or } a \notin \text{Ran}(R) & \text{G3} \\
\gamma^E(a) = \delta \text{ if } a \notin E \text{ and } a \in \text{Ran}(R) & \text{G4} \\
\gamma^E(ax) = \gamma^E(a) \cdot \gamma^E((E - \{a\}) \cup R(a))(x) & \text{G5} \\
\gamma^E(x + y) = \gamma^E(x) + \gamma^E(y) & \text{G6}
\end{array}
\end{array}
\end{align*}
\]
4.5. Example. (i) Suppose $a \parallel d$, $c \parallel b$ (see Figure 6(a)). Then

\[
\gamma^\emptyset(ab || cd) = \gamma^\emptyset(a(b || cd)) + \gamma^\emptyset(c(d || ab)) = \hat{a}^\emptyset d(b || cd) + \ldots = \\
\hat{a}^\emptyset d(bcd + c(d || b)) + \ldots = \hat{a}(\delta + \hat{c}^\emptyset d(bd + bd)) + \ldots = \\
\hat{a}c(ab + bd) + \hat{c}(bd + bd) = (\hat{a} || \hat{c})(\hat{b} || \hat{a}).
\]

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

![Figure 6](image-url)

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 $a^*b = \gamma^\emptyset(a || b)$ for $a \parallel b$.)

(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 $a \parallel c$ and $d \parallel b$. Then

\[
\gamma^\emptyset(X || Y) = \gamma^\emptyset(a(bX || Y) + c(dY || X)) = \hat{a}^\emptyset c(bX || Y) + \delta = \\
\hat{a}^\emptyset c(b(X || Y) + c(dY || bX)) = \hat{a}(\delta + \hat{c}^\emptyset d(bX || bX)) = \\
\hat{a}c(abY)(b(X || Y) + \delta) = \hat{a}c\hat{a}Y(bX || bX) = \\
\hat{a}c\hat{a}Y(X || Y).
\]

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

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 $b || c$, then

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

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
\[ \mu^0_c(c+d \cdot c+d \cdot c+d \cdot c+d) = c \downarrow d \cdot c \uparrow d \cdot c \downarrow d \cdot c \uparrow d. \]

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 Gl-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 \( a_1, \ldots, a_n \):

\[ a_1, \ldots, a_n \parallel b, \]

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 \( \mathcal{H}, \mu_0, \mu_\text{Min} \) in ACP, \( \mu_c, \mu_c^M \) in the mail mechanisms of Section 3 and the present \( \gamma^E \) 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:

- \text{CM} command module
- \text{P} printer
- \text{D} display

The only command that \text{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 \( b \), \( \overline{b} \) we will write, respectively, \( b \) and \( \overline{b} \). So italicized actions are \( \overline{b} \) (not yet completed) and completed actions \( b \) are in usual print.

**CM:**

\[ \text{CM}_0 \quad \text{PC} \quad \text{CM}_1 \quad \text{RP} \quad \text{CM}_2 \]

Figure 7(a).

**P:**

\[ \text{P}_0 \quad \text{STP} \quad \text{P}_1 \quad \text{PAD} \quad \text{P}_2 \quad \text{STOP} \quad \text{POP} \quad \text{P}_3 \quad \text{NP} \]

Figure 7(b).

**D:**

\[ \text{D}_0 \quad \text{D}_{\text{start}} \quad \text{D}_{\text{stop}} \quad \text{D}_{\text{pop}} \quad \text{D}_2 \quad \text{D}_{\text{OK}} \]

Figure 7(c).
In these diagrams, fat arrows represent actions; the other lines identify control points and have no direction.

The causal relations ε R are listed below:

\[
\begin{align*}
PC & \models STP & STP & \models D_{\text{start}} \\
STOP & \models D_{\text{stop}} & D_{\text{stop}} & \models RP \\
POP & \models D_{\text{pop}} & D_{\text{pop}} & \models NP \\
NP & \models D_{\text{OK}}
\end{align*}
\]

The entire system \( S \) is now described by:

\[
S = \mathcal{G}(CM_0 \parallel P_0 \parallel D_0).
\]

Further, let

\[
S^* = \mathcal{G}(NP(CM_1 \parallel P_3 \parallel D_2)),
\]

then it can easily be shown that \( S \) and \( S^* \) satisfy the following recursion equations:

\[
\begin{align*}
S &= PC \cdot STP \cdot \{D_{\text{start}} \cdot PAD \cdot \{STOP \cdot D_{\text{stop}} \cdot RP \cdot S + POP \cdot D_{\text{pop}} \cdot S^*\} + \\
&\quad + PAD \cdot \{D_{\text{start}} \cdot (STOP \cdot D_{\text{stop}} \cdot RP \cdot S + POP \cdot D_{\text{pop}} \cdot S^*) + \\
&\quad\quad + STOP \cdot D_{\text{stop}} \cdot RP \cdot S + POP \cdot D_{\text{pop}} \cdot S^*\}\} \\
S^* &= NP \cdot \{STOP \cdot D_{\text{OK}} \cdot D_{\text{stop}} \cdot RP \cdot S + POP \cdot D_{\text{OK}} \cdot D_{\text{pop}} \cdot S^* + \\
&\quad + D_{\text{OK}} \cdot (STOP \cdot D_{\text{stop}} \cdot RP \cdot S + POP \cdot D_{\text{pop}} \cdot S^*)\}
\end{align*}
\]

REFERENCES


Processes and the denotational semantics of concurrency,
Information and Control, Vol.54, No.1/2, p.70-120, 1982.

Een analyse van de case-study HuP,

A process algebra for the operational semantics of static data flow networks,

[7] BERGSTRA, J.A. & J.W. KLOP,
Process algebra for communication and mutual exclusion. Revised version,

[8] BERGSTRA, J.A., KLOP, J.W. & J.V. TUCKER,
Algebraic tools for system construction,

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

[10] BROCK, J.D. & W.B. ACKERMAN,
Scenarios: A model of non-determinate computation,

[11] BROOKES, S.D.,
On the relationship of CCS and CSP,

[12] BROOKES, S.D. & W.C. ROUNDS,
Behavioural equivalence relations induced by programming logics,

[13] EBERGEN, J.,
On VLSI design,

[14] HENNESSY, M.,
A term model for synchronous processes,

[15] HOARE, C.A.R.,
Communicating Sequential Processes,

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

[17] HOARE, C., BROOKES, S. & W. ROSCOE,  
A theory of communicating sequential processes,  

[18] KAHN, G.,  
The semantics of a simple language for parallel programming,  

[19] KOYMANS, R., VYTOPIL, J. & W.P. DE ROEVER,  
Real-time programming and asynchronous message passing,  

[20] KUIPER, R. & W.P. DE ROEVER,  
Fairness assumptions for CSP in a temporal logic framework,  

[21] LAMPORT, L.  
'Sometime' is sometimes 'NOT never',  

[22] MILNE, G.J.,  
CIRCAL: A calculus for circuit description,  

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

[24] MILNER, R.,  
Calculi for synchrony and asynchrony,  


[26] PNUELI, A.,  
The temporal logic of programs,  

[27] REM, M.,  
Partially ordered computations, with applications to VLSI design,  

[28] DE SIMONE, R.,  
On MEIJE and SCCS: infinite sum operators vs. non-guarded definitions,  

[29] VAN DE SNEPSCHEUT, J.L.A.,  
Trace Theory and VLSI Design,  
[30] WINSKEL, G.,
Event structure semantics for CCS and related languages,

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