Algebra of Broadcasting Systems: Value Passing, Sequential Composition, and Fork

Martin Weichert[*†]

December 5, 1994

Abstract

This paper presents ACBS&, Algebra of Broadcasting Systems with Fork, a process calculus with broadcast communication, which combines value passing and sequential composition.

Putting value passing into a calculus with sequential composition turns out to be more complicated than with a prefixing calculus. A semantics for such a calculus can be defined using environments, as in imperative languages.

Parallel processes have parallel environments, which can lead to ambiguity *which* of the environments shall be passed on to the sequentially following process. ACBS& solves this problem by distinguishing a foreground and a background process. The resulting operator can be interpreted as a *fork* operator, which "forks off" a background process.

The calculus is presented both in a "pure" and a value passing version, together with a complete axiomatisation for the pure calculus.

1 Introduction

The basic ingredients of the calculus ACBS& are:

- CBS with its basic idea broadcast communication, as presented in [Pra91, Pra92],
- first the "pure" and then the value passing version,

into which I have incorporated two ideas from the Dutch work on process algebra, like ACP as presented in [BW90]:

- sequential composition, as opposed to prefixing; with the consequence of having two different actionless processes $\mathbf{o}$ and $\mathbf{1}$, and
- parallel composition is split up into the three operators full merge, left merge, and communication merge; which facilitates axiomatisation.

Parallel composition, which is still present in the equational reasoning, has in the calculus itself been replaced by a *fork* operator, which "forks off" a background process.

[*] *Author's address:* Department of Computer Sciences, Chalmers University of Technology, S-412 96 Gothenburg, Sweden. *E-mail:* `martinw@cs.chalmers.se`

[†] This work was supported by the Esprit Basic Research Action CONCUR2

1.1 What is broadcasting?

In broadcasting communication as presented in [Pra91, Pra92], sending is autonomous and receiving is controlled. Any participant that is ready to speak can do so, while the listener(s) (of which there may be zero, one or many) have to wait until they hear something. The symmetric synchronisation of the handshake model is replaced by an asymmetric synchronisation: the listener is dependent on the speaker, but not the other way round.

The broadcasting model is based on local area networks as well as human speech, and gives rise to a number of interesting programming examples [Pra93].

1.2 Why prefer sequential composition over prefixing?

It is natural to think of events happening sequentially in time. You can express the notion of "repeating a process several times", or even "forever". Furthermore, it emphasizes the role of simple one-action-processes as the building blocks of bigger ones, thus enabling bottom-up process construction.

ACP and related calculi with sequential composition offer several models and axiom systems which are complete for those models. The left merge operator can helpturning an infinite axiom system into a finite one. But their communication primitive seems is predominantly handshake, due to the strong influence of CCS. Applying the ideas of ACP to broadcasting communication instead of handshaking has not been studied yet to my knowledge. This is the purpose of introducing ACBS.

The reader is assumed to be familiar with CCS-like calculi.

2 Discussion of the language constructs

The language presented here consists of a few, simple constructions. Several extensions with more operators and other concepts are possible - some of them face exactly the same problems as existing calculi, others pose quite unexpected new questions. A few of those will be discussed in section 5.

2.1 Simple Actions

The building blocks of processes are simple actions. There are two kinds of actions: send $a!$ and receive $a?$, where a is the actual signal being transmitted. Their meaning is the same as in CBS.

Unlike CBS, an action here is itself a process: The process $a!$ broadcasts the signal a into the environment, and then it terminates. It will ignore any incoming signals, and it does not depend on any other process(es) being ready to receive the signal. Sending is autonomous. The process $a?$ waits until it hears the signal a being broadcast, and then terminates. It cannot ignore a when it comes, but it will ignore anything else. Receiving is dependent on sending.

With value passing, the process $x?$, which can receive *any* value (represented by the variable x), is different from the process $a?$, receiving a particular signal a.

The special signal τ represents noise, a "signal" that carries no information. It is different from the other signals in that it cannot be received.

2.2 Choice

$P + Q$ can behave either as P, or as Q. *Eg*, the process $a! + b?$ can either send an a or receive a b sent from some other process.

2.3 Empty processes

In contrast to prefix-based calculi like CCS and CBS, the use of sequential composition gives rise to *two* empty processes:

2.3.1 Termination

$\mathrm{1}$ does no action, but terminates successfully. For any process P, $\mathrm{1} \cdot P$ is the same as P. It is called ε in [BW90].

2.3.2 Inaction

$\mathbf{o}$ does nothing at all, and prevents anything else from happening afterwards. For any process P, $\mathbf{o} \cdot P$ is the same as $\mathbf{o}$. Is called δ or *deadlock* in [BW90].

2.4 Sequential Composition

$P \cdot Q$ stands for "first P, then Q". If and when P has successfully terminated, Q can happen – like ";" in imperative programming languages.

Note that the condition "P has terminated" is not the same "P can do no more actions". The latter is also true if P just aborts or deadlocks, in which case Q will never start.

In connection with the concept of "forking" processes, the meaning of "termination" itself changes: Process P in $P \cdot Q$ may have started some background process which is still alive when Q is allowed to start executing. Q will then run in parallel with what remains of process P. Instead of "termination" one should better talk about "passing control" from P to Q.

2.5 Parallel Composition. Fork

$P \| Q$ represents the two processes P and Q running in parallel. They can communicate via broadcast. Any of several processes in parallel that is ready to speak can do so, autonomously, while those processes that are ready to listen hear the message, and all others ignore it. $P \| Q$ terminates if and when both P and Q have terminated.

This calculus uses a variant of parallel, the *fork* operator.

$P \& Q$ is the process which first *forks* off process P, and then starts Q, from where on they both run in parallel, possibly communicating with each other. $P \& Q$ behaves exactly as $P \| Q$, except for the passing of control to a process R sequentially composed after it: Whereas in $(P \| Q) \cdot R$ both P and Q have to terminate to let R start, in $(P\&Q)\cdot R$ the process R waits only for termination of Q. The latter expression is equivalent to $P\&(Q\cdot R)$.

The symbol & is chosen because of its use in UNIX shell scripts, where it has the same meaning. It can be defined either as a unary or a binary operator. Both of them are interdefinable in the presence of the other operators of this calculus: $P\&$ can be defined as $P\&\mathbf{1}$, and $P\&Q$ as $(P\&)\cdot Q$.

3 Pure ACBS&: The Syntax and Operational Semantics

3.1 Syntax

Process expressions in "pure" ACBS&, *ie* without value passing, are built using the following syntax:

$$Act ::= a? \mid a! \mid \tau!$$

$$P ::= \mathbf{o} \mid \mathbf{1} \mid Act \mid P+P \mid P\cdot P \mid P\&P$$

The set of all process expressions is called $\mathcal{P}$.

Let $\mathcal{D}$ be a set of data values used as "signals", and $\mathcal{D}_\tau = \mathcal{D}\cup\{\tau\}$. a, b range over signals, including τ where appropriate.

An action consists of a signal together with a punctuation symbol: $a!$ "send a", $a?$ "receive a", or a: "ignore a". The latter is only part of the semantics, given below, it does not appear in the syntax. It is defined in such a way that "P ignores a" *iff* "P does not receive a".

In the following, $\sharp$ stands for !, ?, or :, $\natural$ stands for ! or ?, $\flat$ stands for ? or :.

3.2 Operational Semantics

The operational semantics of ACBS is given by a labelled transition relation on process expressions. There are two kinds of transitions:

$P \xrightarrow{a\natural} P'$ is to be read as "process P does action $a\natural$ and becomes P'".

$P \xrightarrow{\surd} P'$ is to be read as "process P passes on control to the next process in sequential composition, with a "residual" process P' remaining in the background." There is an important special case where P' is an inactive process, like $\mathbf{o}$. A transition like $P \xrightarrow{\surd} \mathbf{o}$ can be read simply as "P terminates".

The semantics is presented in Table 1.

4 Bisimulation and Axiomatisation

Our goal is to find a set of rules for equational reasoning about the calculus, which characterise the desired equivalence between processes, the bisimularity (bisimulation equivalence).

First we define bisimulation via the operational semantics, then we present a set of rules which are sound and complete w r t bisimulation.

Operator	Send	Receive	Ignore[1]	Pass Control
1, Termination				$1 \xrightarrow{\surd} \mathbf{o}$
o, Inaction			$\mathbf{o} \xrightarrow{a:}$	
?, Listen		$a? \xrightarrow{a?} 1$	$a? \xrightarrow{b:}$ (if $a \neq b$)	
!, Send	$a! \xrightarrow{a!} 1$		$a! \xrightarrow{b:}$	
+, Choice ($i \in \{1,2\}$)	$\dfrac{P_i \xrightarrow{a\natural} P'}{P_1 + P_2 \xrightarrow{a\natural} P'}$		$\dfrac{P_1 \xrightarrow{a:} \quad P_2 \xrightarrow{a:}}{P_1 + P_2 \xrightarrow{a:}}$	$\dfrac{P_i \xrightarrow{\surd} P'}{P_1 + P_2 \xrightarrow{\surd} P'}$
$\cdot$, Sequential Composition	$\dfrac{P \xrightarrow{a\sharp} P'}{P \cdot Q \xrightarrow{a\sharp} P' \cdot Q}$			
”	$\dfrac{P \xrightarrow{\surd} P' \quad P'\&Q \xrightarrow{\alpha} R}{P \cdot Q \xrightarrow{\alpha} R}$			
&, fork[2]	$\dfrac{P \xrightarrow{a\sharp_1} P' \; Q \xrightarrow{a\sharp_2} Q'}{P\&Q \xrightarrow{a(\sharp_1 \bullet \sharp_2)} P'\&Q'} \quad (\sharp_1 \bullet \sharp_2 \neq \bot) \quad \begin{array}{c\|ccc} \bullet & ! & ? & : \\ \hline ! & \bot & ! & ! \\ ? & ! & ? & ? \\ : & ! & ? & : \end{array}$			
”	$\dfrac{P \xrightarrow{\surd} P' \quad P'\&Q \xrightarrow{a\natural} R}{P\&Q \xrightarrow{a\natural} R}$		$\dfrac{P \xrightarrow{\surd} P' \quad P'\&Q \xrightarrow{a:}}{P\&Q \xrightarrow{a:}}$	
”				$\dfrac{Q \xrightarrow{\surd} Q'}{P\&Q \xrightarrow{\surd} P\&Q'}$

[1] $P \xrightarrow{a:}$ is an abbreviation for $P \xrightarrow{a:} P$

[2] $\bot$ means “undefined” in the communication algebra (right) for $\bullet$

Table 1: Transition rules for the “pure” calculus

4.1 Strong Bisimulation

(Bi)simulation is defined as usual.[1]

Definition 1 *[Simulation, Bisimulation]*
$\mathcal{R} \subseteq \mathcal{P} \times \mathcal{P}$ *is a simulation if* $\mathcal{R}^\top \cdot \xrightarrow{a\sharp} \subseteq \xrightarrow{a\sharp} \cdot \mathcal{R}^\top$ *for all* $a\sharp$.
$\mathcal{R}$ *is a bisimulation if both* $\mathcal{R}$ *and* $\mathcal{R}^\top$ *are simulations.*

As in CCS/CBS, the largest bisimulation[2] is an equivalence, denoted by $\sim$.

Proposition 2 $\sim$ *is a congruence.*

4.2 Laws

The following laws make use of some auxiliary operators, which are not mentioned in the syntax given above. The purpose of these operators is not to build processes, they only appear in the reasoning about processes.

The process $P\&Q$ is identical to $P\|Q$, except for their termination behavior: $P\&Q$ terminates if and when Q does. Termination of the "background process" P has no effect at all. Therefore we have the equality:

$$P\&Q = (P \cdot \mathbf{o})\|Q$$

The three operators full merge $\|$, left merge $\mathbin{⫽}$, and communication merge $\#$ are taken from ACP [BW90]. We use the symbol $\#$ for communication merge; in ACP it is written $|$.

$P\|Q$ has in general three possibilities for its first action: Either P can do an action (which Q ignores), or Q does an action (which P ignores), or both of them engage in a communication action between them. These three possibilities are reflected by the important equality:

$$P\|Q = P \mathbin{⫽} Q + Q \mathbin{⫽} P + P \# Q$$

$P \# Q$ must start with a communication action between P and Q. $P \mathbin{⫽} Q$ must start with an action from P, which Q ignores. If Q queries for a signal a, then it *must* listen to it when it comes - it cannot ignore it. Thus, the process $P \mathbin{⫽} Q$ *cannot* do an a-action if Q queries for a.

The actions of $P \mathbin{⫽} Q$ are therefore restricted by the "query set" of process Q, *ie* the set of all labels a such that Q can start with an $a?$-transition, written as $Q^? = \{a \colon \exists Q'. \, Q \xrightarrow{a?} Q'\}$.

The restricted action $a\natural \backslash Q^?$ is the same as $a\natural$ *iff* a is not in the set $Q^?$; otherwise it is equal to $\mathbf{o}$.

The laws for strong bisimulation are given in Table 2.

---

[1] I prefer the concise formulation $\mathcal{R}^\top \cdot \xrightarrow{a\sharp} \subseteq \xrightarrow{a\sharp} \cdot \mathcal{R}^\top$, or, using the relational division operator, $\mathcal{R}^\top \subseteq \xrightarrow{a\sharp} \cdot \mathcal{R}^\top / \xrightarrow{a\sharp}$, over the usual one, which is equivalent:
"whenever $P\mathcal{R}Q$, if $P \xrightarrow{a\sharp} P'$ then $\exists Q'$ such that $Q \xrightarrow{a\sharp} Q'$ and $P'\mathcal{R}Q'$ ".
$\mathcal{R}^\top$ is the converse (or "transpose") of $\mathcal{R}$.

[2] "Bisimulation" in this paper always means "strong", unless explicitly stated that it is "weak".

$$
\begin{array}{rrcl}
(+\textsc{com}) & P+Q & = & Q+P \\
(+\textsc{ass}) & (P+Q)+R & = & P+(Q+R) \\
(+\textsc{idp}) & P+P & = & P \\
(+\mathbf{o}) & P+\mathbf{o} & = & P \\
 & & & \\
(\mathbf{o}\,\cdot) & \mathbf{o}\cdot P & = & \mathbf{o} \\
(\mathbf{1}\,\cdot) & \mathbf{1}\cdot P & = & P \\
(\,\cdot\,\textsc{ass}) & (P\cdot Q)\cdot R & = & P\cdot(Q\cdot R) \\
(+\,\cdot) & (P_1+P_2)\cdot Q & = & P_1\cdot Q+P_2\cdot Q \\
(\textit{but:} & P\cdot(Q_1+Q_2) & \neq & P\cdot Q_1+P\cdot Q_2\ !) \\
 & & & \\
(\&\|) & P\&Q & = & (P\cdot\mathbf{o})\|Q \\
 & & & \\
(\textsc{exp}) & P\|Q & = & P\lfloor\!\lfloor Q+Q\lfloor\!\lfloor P+P\#Q \\
 & & & \\
(\mathbf{o}\lfloor\!\lfloor) & \mathbf{o}\lfloor\!\lfloor Q & = & \mathbf{o} \\
(+\lfloor\!\lfloor) & (P_1+P_2)\lfloor\!\lfloor Q & = & P_1\lfloor\!\lfloor Q+P_2\lfloor\!\lfloor Q \\
(\natural\lfloor\!\lfloor) & a\natural\lfloor\!\lfloor Q & = & a\natural\setminus Q^?\cdot Q \\
(\,\cdot\lfloor\!\lfloor) & (a\natural\cdot P)\lfloor\!\lfloor Q & = & a\natural\setminus Q^?\cdot(P\|Q) \\
 & & & \\
(\#\textsc{com}) & P\#Q & = & Q\#P \\
(\mathbf{o}\#) & \mathbf{o}\#Q & = & \mathbf{o} \\
(+\#) & (P_1+P_2)\#Q & = & P_1\#Q+P_2\#Q \\
(\natural\,\#) & a\natural\#(b\natural'\cdot Q) & = & (a\natural\#b\natural')\cdot Q \\
(\,\cdot\#) & (a\natural\cdot P)\#(b\natural'\cdot Q) & = & (a\natural\#b\natural')\cdot(P\|Q) \\
 & & & \\
(!\#!) & a!\#a! & = & \mathbf{o} \\
(!\#?) & a!\#a? & = & a! \\
(?\#?) & a?\#a? & = & a? \\
(\#\neq) & a\natural\#b\natural' & = & \mathbf{o},\text{ if } a\neq b \\
 & & & \\
(\setminus\in) & a\natural\setminus Q^? & = & \mathbf{o},\text{ if } a\in Q^? \\
(\setminus\notin) & a\natural\setminus Q^? & = & a\natural,\text{ if } a\notin Q^? \\
 & & & \\
(\mathbf{o}^?) & \mathbf{o}^? & = & \emptyset \\
(!^?) & (a!)^? & = & \emptyset \\
(?^?) & (a?)^? & = & \{a\} \\
(+^?) & (P+Q)^? & = & P^?\cup Q^? \\
(\,\cdot^?) & (P\cdot Q)^? & = & P^?
\end{array}
$$

Table 2: Laws for strong bisimulation

Let

$$
\begin{aligned}
\mathcal{L}aws \quad = \{ \quad & (+\textsc{com}), (+\textsc{ass}), (+\textsc{idp}), (+\mathbf{o}), (\mathbf{o}\,\cdot\,), (1\,\cdot\,), (+\,\cdot\,), (\,\cdot\,\textsc{ass}), (\&\|), (\textsc{exp}),\\
& (\mathbf{o}\mathbb{L}), (+\mathbb{L}), (\natural\,\mathbb{L}), (\,\cdot\,\mathbb{L}), (\text{⫲}\textsc{com}), (\mathbf{o}\text{⫲}), (+\text{⫲}), (\natural\,\text{⫲}), (\,\cdot\,\text{⫲}),\\
& (!\text{⫲}!), (!\text{⫲}?), (?\text{⫲}?), (\text{⫲}\neq), (\backslash \in), (\backslash \notin), (\mathbf{o}^?), (!\,^?), (?\,^?), (+^?), (\,\cdot\,^?) \;\}
\end{aligned}
$$

Proposition 3 *All of the laws are sound w r t $\sim$.*

To prove *eg* that the law $(\,\cdot\,\textsc{ass})$ is sound w r t $\sim$, I show that the relation

$$\mathcal{R} \stackrel{def}{=} \{\langle (P\cdot Q)\cdot R, P\cdot(Q\cdot R)\rangle : P, Q, R \in \mathcal{P}\} \cup Id_{\mathcal{P}}$$

is a bisimulation.

(The relation

$$\mathcal{R}' \stackrel{def}{=} \{\langle (P\cdot Q)\cdot R, P\cdot(Q\cdot R)\rangle : P, Q, R \in \mathcal{P}\}$$

is not enough: For some P, Q and R, both $(P\cdot Q)\cdot R$ and $P\cdot(Q\cdot R)$ may have a transition to the *same* process, rather than to two processes which are in the relation $\mathcal{R}'$.)

The soundness proofs can be machine checked. So far, I have translated the syntax and semantics of the calculus into the logical framework ALF [MN94], and have proved the soundness of laws like $1\cdot P \sim \mathbf{o}\&P$ and $\mathbf{o}\&P \sim P$.

4.3 Basic Terms

A process expression is called a *basic term* if it is in the following form:

$$B = \sum_i a_i\natural_i + \sum_j a_j\natural_j \cdot B_j$$

where the B_j themselves are basic terms. An empty sum stands for $\mathbf{o}$.

4.4 Completeness

The above laws can be turned into a Term Rewriting System, by reading all the equal signs as arrows from left to right. That TRS reduces every process to a basic term:

Proposition 4 *For every process $P \in \mathcal{P}$, there is a basic term $B \in \mathcal{P}$, such that*

$$\mathcal{L}aws \vdash P = B$$

The proof works by induction over the structure of basic terms.

Basic terms that are bisimilar must have the same (*ie*, bisimilar) summands (modulo commutativity, associativity, and idempotence of $+$ and occurrence of $\mathbf{o}$).

Let

$$\mathcal{L}aws^+ = \{(+\textsc{com}), (+\textsc{ass}), (+\textsc{idp}), (+\mathbf{o})\}$$

Proposition 5 *For all basic terms $B_1, B_2 \in \mathcal{P}$: if $B_1 {\sim} B_2$ then*

$$\mathcal{L}aws^+ \vdash B_1 = B_2$$

Thus,

Proposition 6 *The set $\mathcal{L}aws$ is complete w r t bisimulation.*

4.5 More laws

Some other interesting equalities, which are not needed for the completeness result, are stated below.

$$\begin{array}{rrcl}
(\|\mathbf{o}) & P\|\mathbf{o} & = & P\cdot\mathbf{o} \\
(\|\textsc{com}) & P\|Q & = & Q\|P \\
(\|\textsc{ass}) & (P\|Q)\|R & = & P\|(Q\|R)
\end{array}$$

4.6 Expansion Theorem

CCS-like calculi usually have an axiomatisation including an "Expansion Theorem", which is really not an axiom but an axiom schema for infinitely many axioms. Note that the list above does not contain any explicit Expansion Theorem. Instead it has the equality ($\textbf{EXP}$), which together with the distributivity laws for $\lfloor\!\lfloor$ and $\#$ plays a similar role, thus making a *finite* axiomatisation possible. Faron Moller showed in his thesis [Mol89], that the $\lfloor\!\lfloor$ operator is necessary for finite axiomatisation.

5 Extensions

5.1 Recursion

So far, all processes have finite transition trees. There are various ways the calculus can be extended to include some form of recursion and thus processes that have infinite transition trees.

The most general form of recursion allows a process to contain itself within any operator, *eg*

$$X = a! + b! \cdot X \cdot c!$$

$$Y = a! \cdot (Y\|b!)$$

Both processes X and Y have infinitely many *different* states in their transition trees.

If however recursion is restricted such that the recursive appearance of the process is not allowed within a parallel, and only on the right side of a sequential, then the process can only reach a finite number of different states.

The first form of recursion therefore appears to be more expressive. This difference in expressivity reminds of the one between context-free (Chomsky 2) and regular (Chomsky 3) languages.

5.1.1 Loop

A simple form of recursion is made possible by the sequential composition, P^ω, "repeat P forever". It has the semantics rule:

$$\frac{P \xrightarrow{a\sharp} P'}{P^\omega \xrightarrow{a\sharp} P' \cdot P^\omega}$$

and the following laws:

$$\begin{array}{rrcl} (\textsc{unfold}) & P^\omega & = & P \cdot P^\omega \\ (\textsc{fold}) & \text{if } X & = & P \cdot X, \text{ then } X = P^\omega \end{array}$$

5.1.2 Loop and Fork

Suppose we want to define a generator process Gen which, every time it receives a signal gen, creates a new copy of some process P to run in parallel with Gen itself. Gen can be written recursively as: $Gen = gen? \cdot (P \| Gen)$. Using the (unary or binary) fork operator, this is equivalent to $Gen = gen? \cdot (P \& Gen)$ or $Gen = gen? \cdot P\& \cdot Gen$. Thus it can be rewritten with a loop as $Gen = (gen? \cdot P\&)^\omega$. Thus, the loop operator as a "restricted" form of recursion is not so restricted after all. It is not clear whether this rewriting offers any advantage or only shifts the complexity to a different operator.

6 Discussion: Value Passing and Environments

6.1 Why environments? Prefixing vs. Sequential Composition

We extend the calculus with value passing.

Process expressions can be built with the following actions:

$$Act ::= \tau! \mid e! \mid x?$$

where x stands for a variable, and e for any expression, possibly containing variables, yielding a result. The transition system defined in the operational semantics, however, is still labelled with single actions, not with variables or expressions.

In a prefixing calculus, processes can be defined as:

$$P ::= \mathbf{o} \mid Act.\, P \mid P + P \mid P \| P$$

Action $x?$ *binds* the variable name x, which may occur free in e. In $x!\,.\,P$, x is free, in $x?\,.\,x!\,.\,P$ it is bound.

With sequential composition, processes look like this:

$$P ::= \mathbf{o} \mid Act \mid P \cdot P \mid P + P \mid P \| P$$

$x?$ cannot be seen as a variable binding here, if we want $\cdot$ to be associative: The scope of x in $(x? \cdot P) \cdot Q$ would be P, but in $x? \cdot (P \cdot Q)$ it would be $P \cdot Q$.

Now we want to give meaning to processes like $x? \cdot x!$, by separately interpreting its parts $x?$ and $x!$ and combining them. In a prefixing calculus like CBS one can express value passing with a rule

$$x?\,.\,P \xrightarrow{v?} P[v/x]$$

(note that the two terms on both sides of the arrow have no free variables, though P itself does!). We have no directly corresponding rule here. If we try to define

$$x? \xrightarrow{v?} 1$$

we find that the information about the actually transmitted value v is lost. The prefixing calculus does not need to give any semantics to expressions with free variables, but we have to. By introducing *environments*, assignments of variable names to values, the information can be conveyed from one process to its successor. Receiving a value changes the environment:

$$\langle x?\,,\rho\rangle \xrightarrow{v?} \langle 1, \rho[v/x]\rangle$$

Sending a value uses the information stored in the environment:

$$\langle e!\,,\rho\rangle \xrightarrow{(\llbracket e \rrbracket \rho)!} \langle 1, \rho\rangle$$

The operational semantics is no longer a transition system between processes, but between processes augmented with environments:

$$\rightarrow \quad \subseteq \; (\bar{\mathcal{P}} \times \mathcal{E}nv) \times Act \times (\bar{\mathcal{P}} \times \mathcal{E}nv)$$

Defining an operational semantics is more difficult, and also more imperative-style. There is no value passing in ACP as presented in [BW90].

6.2 Environments of parallel processes

Each process running in parallel has its own environment, as given by the rule:

$$\frac{\langle P,\rho\rangle \xrightarrow{a\sharp_1} \langle P',\rho'\rangle \quad \langle Q,\sigma\rangle \xrightarrow{a\sharp_2} \langle Q',\sigma'\rangle}{\langle P,\rho\rangle \| \langle Q,\sigma\rangle \xrightarrow{a\sharp_1 \bullet \sharp_2} \langle P',\rho'\rangle \| \langle Q',\sigma'\rangle}$$

This approach makes the variables local to each parallel component. It works fine for static systems, where the number of parallel processes stays constant all the time.

If we want to have dynamic systems, where processes can split into two, or join into one, we need some more rules.

6.3 Splitting processes

A process splits into two in the expression $P \cdot (Q \| R)$. Both Q and R start with the same environment that P ended up with. This will be made formal later, using augmented processes and configurations.

6.4 Joining processes

In $(P\|Q)\cdot R$, two processes join into one. When P and Q terminate with different environments ρ and σ, which environment should R inherit? (P and Q could for example perform two parts of a computation in parallel, the final result of which shall then be used by R.)

There are several possibilities.

6.4.1 Ranking

Out of several parallel components, one is declared as "main", and its environment will be inherited.

6.4.2 Random

R inherits either ρ or σ by non-deterministic choice.

6.4.3 Combine

ρ and σ are somehow merged into one environment $\rho \oplus \sigma$. A restriction on P and Q could be introduced that shared variables must be read-only.

6.4.4 Agree

P and Q both give a result; and the results given by both must agree. (*cf* LOTOS).

6.4.5 Nothing

R gets no information out of P and Q. The scope of any variables occurring in P and Q is limited to these.

6.4.6 Disallow

Joining of processes if not allowed. The first operand of a sequential composition must not contain any $\|$.

6.5 Fork

The problem with the joining processes is partly due to the symmetry of the parallel operator. In $(P\|Q)\cdot R$, R has to wait for both P and Q to terminate, and we don't know from which of them R should inherit the environment. In contrast, in $(P\&Q)\cdot R$, R waits only for Q to terminate, so it seems natural to let R inherit the environment of Q.

7 ACBS& with Value Passing: The Syntax and Operational Semantics

After all these theoretic considerations, here is the calculus with value passing.

7.1 Syntax

ACBS& process expressions are built using the following syntax:

$$P ::= \mathbf{o} \;\Big|\; 1 \;\Big|\; x? \;\Big|\; e! \;\Big|\; \tau! \;\Big|\; P+P \;\Big|\; P\cdot P \;\Big|\; P\&P$$

7.2 Operational Semantics

For the operational semantics with value passing we need some auxiliary definitions.

Processes are *augmented* with *environments* in certain positions.

The syntax for augmented processes is given by:

$$\bar{P} ::= \mathbf{o} \;\Big|\; 1 \;\Big|\; x? \;\Big|\; e! \;\Big|\; \tau! \;\Big|\; \bar{P}+\bar{P} \;\Big|\; \bar{P}\cdot P \;\Big|\; \bar{P},\rho\&\bar{P}$$

where P stands for a process, $\bar{P}$ for an augmented process, and ρ for an environment. Note that the second component of a sequential composition is *not* augmented with an environment - it will inherit the one from its predecessor when that passes over control.

We will call a pair consisting of an *augmented process* and an *environment*, such as $\bar{P}, \rho$ (dropping the pairing parentheses), a *configuration*.

Augmenting a process P with an *environment* ρ is defined as follows:

$$\begin{array}{lcl}
\mathbf{o}^{\rho} & = & \mathbf{o} \\
1^{\rho} & = & 1 \\
x?^{\rho} & = & x? \\
e!^{\rho} & = & e! \\
(P_1+P_2)^{\rho} & = & P_1{}^{\rho}+P_2{}^{\rho} \\
(P\cdot Q)^{\rho} & = & P^{\rho}\cdot Q \\
(P\&Q)^{\rho} & = & P^{\rho},\rho\&Q^{\rho}
\end{array}$$

The operational semantics of ACBS with value passing is given by a labelled transition relation between configurations; so the semantics associated to a process P starting in an environment ρ will be the transition system of the configuration P^{ρ}, ρ.

The semantics is presented in Table 3. $\tilde{P}$, $\tilde{P}'$, $\tilde{Q}$... range over configurations.

8 Denotational semantics. Implementation

The calculus can also be given a denotational semantics, where the process domain of transition trees is defined with the help of, *eg*, complete partial orders [Abr91] or metric spaces [vB93]. I have implemented the calculus (with value passing) in **LML**, a lazy functional language, by translating process terms into transition trees which I represent as lazy infinite data structures. In **ALF** [MN94] I have done another implementation, so far only the "pure" calculus, and done some machine-assisted correctness proofs.

Operator	Send	Receive	Ignore[1]	Pass Control
1, Termination				$1, \rho \xrightarrow{\surd} \mathbf{o}, \rho$
$\mathbf{o}$, Inaction			$\mathbf{o}, \rho \xrightarrow{v:}$	
$?$, Listen		$x?\,, \rho \xrightarrow{v?} 1, \rho[v/x]$	$x?\,, \rho \xrightarrow{\tau:}$	
$!$, Send	$e!\,, \rho \xrightarrow{\llbracket e \rrbracket \rho !} 1, \rho$		$e!\,, \rho \xrightarrow{v:}$	
$+$, Choice ($i \in \{1,2\}$)	$\dfrac{P_i, \rho \xrightarrow{a\natural} \tilde{P}'}{P_1 + P_2, \rho \xrightarrow{a\natural} \tilde{P}'}$		$\dfrac{P_1, \rho \xrightarrow{v:} \quad P_2, \rho \xrightarrow{v:}}{P_1 + P_2, \rho \xrightarrow{v:}}$	$\dfrac{P_i, \rho \xrightarrow{\surd} \tilde{P}'}{P_1 + P_2, \rho \xrightarrow{\surd} \tilde{P}'}$
$\cdot$, Sequential Composition	$\dfrac{P, \rho \xrightarrow{a\sharp} P', \rho'}{P \cdot Q, \rho \xrightarrow{a\sharp} P' \cdot Q, \rho'}$			
”	$\dfrac{P, \rho \xrightarrow{\surd} \tilde{P}' \quad \tilde{P}' \& Q^{\rho}, \rho \xrightarrow{\alpha} \tilde{R}}{P \cdot Q, \rho \xrightarrow{\alpha} \tilde{R}}$			
$\&$, fork[2]	$\dfrac{\tilde{P} \xrightarrow{a\sharp_1} \tilde{P}' \quad \tilde{Q} \xrightarrow{a\sharp_2} \tilde{Q}'}{\tilde{P} \& \tilde{Q} \xrightarrow{a(\sharp_1 \bullet \sharp_2)} \tilde{P}' \& \tilde{Q}'} \; (\sharp_1 \bullet \sharp_2 \neq \perp)$		$\begin{array}{c c c c} \bullet & ! & ? & : \\ \hline ! & \perp & ! & ! \\ ? & ! & ? & ? \\ : & ! & ? & : \end{array}$	
”	$\dfrac{\tilde{P} \xrightarrow{\surd} \tilde{P}' \quad \tilde{P}' \& \tilde{Q} \xrightarrow{a\natural} \tilde{R}}{\tilde{P} \& \tilde{Q} \xrightarrow{a\natural} \tilde{R}}$		$\dfrac{\tilde{P} \xrightarrow{\surd} \tilde{P}' \quad \tilde{P}' \& \tilde{Q} \xrightarrow{v:}}{\tilde{P} \& \tilde{Q} \xrightarrow{v:}}$	
”				$\dfrac{\tilde{Q} \xrightarrow{\surd} \tilde{Q}'}{\tilde{P} \& \tilde{Q} \xrightarrow{\surd} \tilde{P} \& \tilde{Q}'}$

[1] $P, \rho \xrightarrow{v:}$ is an abbreviation for $P, \rho \xrightarrow{v:} P, \rho$

[2] $\perp$ means "undefined" in the communication algebra (right) for $\bullet$

Table 3: Transition rules for the value passing calculus

9 Conclusions. Where to go on

9.1 Broadcast vs. Handshake

The work so far shows that broadcast communication can be realised in a sequential setting just as well as handshake communication. Nothing in ACP forces it to use handshaking. The difficulties encountered with value passing in a sequential setting are independent of the underlying communication model.

9.2 Sequential vs. Prefixing

The idea of sequential composition, a more "symmetric", homogeneous operator than prefixing, seems appealing. As long as only the pure calculus is considered, sequential composition works just as well as prefixing. With value passing, however, things start getting difficult, without any obvious advantage of sequential composition to be seen.

The problem of passing values in some context with sequential and parallel composition is however quite a common one and appears in many different formalisms, such as UNIX shell scripts, LOTOS, "Fork Calculus" [Hav92], Concurrent ML, OCCAM, μCRL [Gro91]. The concept of forking seems promising towards a solution.

9.3 Other

A question not yet treated here is how to define a suitable *weak bisimulation* and which properties it will have. We are also looking for more motivating examples. Space didn't permit me to put in here any of the examples I have, unfortunately.

10 Acknowledgements

Special thanks to my supervisor K.V.S. Prasad, without whom this work would not have been possible, to the concurrency group at Chalmers, and Thorsten Altenkirch for discussion and encouragement.

References

[Abr91] S. Abramsky. A domain equation for bisimulation. *Information and Computation*, 92(2):161–218, June 1991.

[BB93] J. C. M. Baeten and J. A. Bergstra. On sequential composition, action prefixes and process prefix. Technical Report P9305, University of Amsterdam, March 1993.

[BBK86] J. C. M. Baeten, J. A. Bergstra, and J. W. Klop. Decidability of bisimulation equivalence for processes generating context-free languages. Technical Report CS-R8632, Centre for Mathematics and Computer Science, Amsterdam, 1986.

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

[CHM93] Søren Christensen, Yoram Hirshfeld, and Faron Moller. Bisimulation equivalence is decidable for all basic parallel processes. In *CONCUR-93*, Hildesheim, 1993.

[Gro91] Jan Friso Groote. *Process Algebra and Structured Operational Semantics*. PhD thesis, Universiteit van Amsterdam, November 1991.

[Hav92] Klaus Havelund. The fork calculus, 1992. presented at the Workshop in Bergen, fall 1992.

[HL92] Matthew Hennessy and Huimin Lin. Symbolic bisimulations. Computer Science Internal Report 1/92, University of Sussex, April 1992.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*. Lecture Notes in Computer Science. Springer-Verlag, 1980.

[MN94] Lena Magnusson and Bengt Nordström. The ALF proof editor and its proof engine. In *Types for Proofs and Programs*, LNCS, pages 213–237, Nijmegen, 1994. Springer-Verlag.

[Mol89] Faron Moller. *Axioms for Concurrency*. PhD thesis, University of Edinburgh, October 1989.

[Pra91] K. V. S. Prasad. A calculus of broadcasting systems. In *TAPSOFT'91 Volume 1: CAAP*, April 1991. Springer Verlag LNCS 493.

[Pra92] K. V. S. Prasad. A calculus of value broadcasts. Technical Report 69, Department of Computer Science, Chalmers University of Technology, October 1992.

[Pra93] K. V. S. Prasad. Programming with broadcasts. Technical Report 74, Department of Computer Science, Chalmers University of Technology, September 1993.

[vB93] Franck van Breugel. Three metric domains of processes for bisimulation. Technical Report CS-R9335, CWI, Amsterdam, 1993.

[Wei93] Martin Weichert. Algebra of communicating systems. In *Wintermötet Tanum Strand*, January 1993. Draft Proceedings.

[Wei94] Martin Weichert. Value passing and sequential composition. In *Proceedings of La Wintermöte '94 Lundsbrunn*, January 1994.