Process algebra with pointers

Process Algebra with Pointers

J.C.M. Baeten[1], J.A. Bergstra[23], and L.M.G. Feijs[14]

[1] Department of Computer Science, Technische Universiteit Eindhoven,
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
{j.c.m.baeten, l.m.g.feijs}@tue.nl
[2] Programming Research Group, Universiteit van Amsterdam,
Kruislaan 403, 1098 SJ Amsterdam, The Netherlands
[3] Department of Philosophy, Universiteit Utrecht,
Heidelberglaan 8, 3584 CS Utrecht, The Netherlands
janb@phil.uu.nl
[4] Department of Industrial Design, Technische Universiteit Eindhoven,
P.O. Box 513, 5600 MB Eindhoven, The Netherlands

Abstract. We present a process algebra for mobile processes without bound or free variables. Instead, pointers are used, that refer back to an action executed in the history of a process. The situation is comparable to a presentation of the λ-calculus with De Bruijn indices.
Note: Report CS-R 02-03, Department of Mathematics and Computer Science, Technische Universiteit Eindhoven,
http://www.win.tue.nl/st/medew/pubbaeten.html.

1 Introduction

For some time now, there is research on process algebras that can deal with mobility of processes and communication links. Most of this research centres around the π-calculus (see [16], a nice introduction is [15]). An essential part of the π-calculus is the use of free and bound variables, different forms of variable binding, α-conversion, scope extrusion. This makes it difficult to understand for some, and makes implementations hard. In this paper, we develop a process algebra that can deal with mobility, that does not use variables. Instead, we use *pointers*, also called history pointers or step counters. A pointer is a natural number n, and refers back to an action that happened n steps ago in the history of the process that is being executed. Such history pointers have come up in the study of process algebras with a refinement operator (stated differently, a process algebra with durational actions), see e.g. [10, 2], dating back to work on causal trees, see [11]. In this paper, we develop a process algebra with pointers in full, show some basic results, and provide some examples that show how mobility of processes and communication links can be handled.

The advantage of our approach is that we are still within the familiar framework of process algebra, and not in a calculus with variable binding constructs. Thus, we can use standard reasoning and techniques. For instance, we can work in the setting of strong bisimulation equivalence, or we can consider a projective

limit model, with finite approximations of infinite behaviour. We can use structured operational semantics, and standard results from this area. A disadvantage is that we do not know how to treat silent steps or forms of weak bisimulation equivalence in our setting.

The introduction of numbers in the place of variables is comparable to a presentation of the λ-calculus with De Bruijn indices or nameless dummies, see [9]. Also there, numbers are used that point back in a term (or tree). These indices were introduced in 1972 for the automatic manipulation of terms needed in the implementation of the proof checker Automath (see [17], this statement is quoted from [5], page 579). Use of free and bound variables remains difficult around α-conversion when doing substitutions, see e.g. [6] page 126. De Bruijn indices still play an important role in the theory and practice of the λ-calculus and functional programming. Some references are [18, 14, 19, 12, 13].

The only reference to the use of De Bruijn indices in the π-calculus we could find is [20], where they were used in the implementation of the Mobility Workbench. However, no theoretical treatment is provided.

1.1 Acknowledgement

The authors gratefully acknowledge the help of Tijn Borghuis (Techn. Univ. Eindhoven).

2 ACP with Pointers

We start out from the well-known theory ACP [7, 4], slightly modified along the lines of [2] in order to deal with pointer updates. The signature of ACP contains the following ingredients:

- A given finite set of action labels L with typical elements $\alpha, \beta, \ldots$.
- *Atomic actions* $\alpha(d_1, \ldots, d_k, \pi i_1, \ldots, \pi i_n)$, abbreviated $\alpha(\vec{d}, \vec{\pi})$ consist of an action label and a number of parameters. First of all, there can be a number of data parameters $d_1, \ldots, d_k$. Then, there can be a number of pointers πi, for a natural number i. In order to distinguish pointers from other numbers that might occur in terms, we precede them by the letter π. Two pointers are the same if they have the same number. Two pointer sequences are equal if they have the same length and consist of the same elements. The set of atomic actions Act is ranged over by letters $a, b, \ldots$.
- A constant *inaction* denoted δ (not an atomic action). This constant is the neutral element of alternative composition. This process disallows termination, so can be used to denote deadlock behaviour. It doesn't take parameters, i.e. for each parameter sequence, $\delta(\vec{d}, \vec{\pi})$ denotes δ.
- A binary operator *alternative composition* or *choice*, denoted $+$. Choice is resolved by the execution of an action.
- A binary operator *sequential composition*, denoted $\cdot$.

- A binary operator *parallel composition* or *merge*, denoted $\|$. In a parallel composition, one of the components executes an action, or more than one component execute an action together, a communication action. The merge is axiomatised using two auxiliary operators: *left merge*, denoted $\lfloor\!\lfloor$, and *communication merge*, denoted $\mid$. Communication on action labels is given by a partial, commutative and associative function γ, considered to be a parameter of the theory. Atomic actions can only communicate if they have the same parameters.
- A unary operator *encapsulation*, denoted ∂_H, that blocks the execution of atomic actions with a label from the parameter set $H \subseteq L$. It is used to encapsulate communicating actions from the environment.
- The *pointer shift* operator Π^+ will update all pointers, when an action is executed in a parallel component. It is axiomatized using auxiliary operators $\Pi^+_{>n}$, for natural numbers n.
- The *pointer forgetting* operator ϕ_I will forget all pointers of actions with labels in $I \subseteq L$.

If action $a \in A$ is of the form $a \equiv \alpha(\vec{d}, \vec{\pi})$, then the label of a is α, expressed as $L(a) = \alpha$. Of all operators, sequential composition binds the strongest, and alternative composition the weakest. The other operators are not ranked. The axioms of ACP are well-known, and given for easy reference in Table 1. In order to save on the number of axioms, we enforce commutativity of communication merge from the start (we have no need to consider models where this does not hold, anyway). We use the presentation and naming of axioms of [8].

$x + y = y + x$	A1	$\alpha(\vec{d}, \vec{\pi}) \mid \beta(\vec{d}, \vec{\pi}) = \zeta(\vec{d}, \vec{\pi})$ if $\gamma(\alpha, \beta) = \zeta$	CF1
$(x + y) + z = x + (y + z)$	A2	$a \mid b = \delta$ otherwise	CF2
$x + x = x$	A3		
$(x + y) \cdot z = x \cdot z + y \cdot z$	A4	$x \,\|\, y = x \lfloor\!\lfloor y + y \lfloor\!\lfloor x + x \mid y$	CM1
$(x \cdot y) \cdot z = x \cdot (y \cdot z)$	A5	$a \lfloor\!\lfloor x = a \cdot \Pi^+(x)$	CM2π
$x + \delta = x$	A6	$(a \cdot x) \lfloor\!\lfloor y = a \cdot (x \,\|\, \Pi^+(y))$	CM3π
$\delta \cdot x = \delta$	A7	$(x + y) \lfloor\!\lfloor z = x \lfloor\!\lfloor z + y \lfloor\!\lfloor z$	CM4
		$x \mid y = y \mid x$	CMC
$\partial_H(a) = \delta$ if $L(a) \in H$	D1π	$(a \cdot x) \mid b = (a \mid b) \cdot x$	CM5
$\partial_H(a) = a$ otherwise	D2π	$(a \cdot x) \mid (b \cdot y) = (a \mid b) \cdot (x \,\|\, y)$	CM7
$\partial_H(x + y) = \partial_H(x) + \partial_H(y)$	D3	$(x + y) \mid z = x \mid z + y \mid z$	CM8
$\partial_H(x \cdot y) = \partial_H(x) \cdot \partial_H(y)$	D4		

Table 1. Axioms of ACP with pointer update ($\alpha, \beta, \zeta \in L$, $H \subseteq L$, $a, b \in A \cup \{\delta\}$).

The main difference with the standard presentation is in the treatment of interleaving: in axioms CM2π CM3π, when an action from the left component is executed, all pointers in the right component that refer to something outside

this component should be incremented. All pointers that refer to something internally are unchanged. Thus, in the initial position all pointers greater than 0 are incremented, in the second position all pointers greater than 1, and so on. These axioms are taken from [2], where the pointer update operator was called *history pointer shift*, which was used in order to deal with durational actions in ST bisimulation semantics. Next, we present axioms for the pointer update operator Π^+ and the pointer forgetting operator ϕ_I in Table 2.

$\Pi^+(x) = \Pi^+_{>0}(x)$	PI0
$\Pi^+_{>n}(\delta) = \delta$	PI1
$\Pi^+_{>n}(\alpha(\vec{d},\vec{\pi})) = \alpha(\vec{d},\Pi^+_{>n}(\vec{\pi}))$	PI2
$\Pi^+_{>n}(\pi i) = \pi i + 1$ if $i > n$	PI3
$\Pi^+_{>n}(\pi i) = \pi i$ if $i \leq n$	PI4
$\Pi^+_{>n}(a \cdot x) = \Pi^+_{>n}(a) \cdot \Pi^+_{>n+1}(x)$	PI5
$\Pi^+_{>n}(x + y) = \Pi^+_{>n}(x) + \Pi^+_{>n}(y)$	PI6
$\phi_I(\alpha(\vec{d},\vec{\pi})) = \alpha(\vec{d})$ if $\alpha \in I$	PH1
$\phi_I(a) = a$ otherwise	PH2
$\phi_I(x + y) = \phi_I(x) + \phi_I(y)$	PH3
$\phi_I(x \cdot y) = \phi_I(x) \cdot \phi_I(y)$	PH4

Table 2. Axioms of pointer update and erase ($\alpha \in L, a \in A \cup \{\delta\}$).

An interesting property we can prove by means of structural induction for all closed terms over the theory of ACP with pointers is the following:

$$\Pi^+_{>n}(x \parallel y) = \Pi^+_{>n}(x) \parallel \Pi^+_{>n}(y).$$

The definition of an operational semantics by means of SOS deduction rules is also standard. By means of these rules, we define binary relations $. \xrightarrow{a} .$ and unary relations $. \xrightarrow{a} \surd$ on closed terms (for $a \in A$). Intuitively, they have the following meaning:

- $x \xrightarrow{a} x'$ means that x evolves into x' by executing a
- $x \xrightarrow{a} \surd$ means that x successfully terminates upon execution of a

We present the rules in Table 3. Rules for pointer update and pointer erase only change something inside atomic actions, so are not very interesting and left out. The rules for communication in lines 6 and 7 only hold in case $a \mid b$ is defined to be an atomic action, so not equal to δ. The rules are in the so-called *path* format, see e.g. [3], from which we know that the semantics induced by the rules have some nice properties.

We define (strong) bisimulation equivalence in the standard way, based on these rules. Since the rules are in *path* format, bisimulation is a congruence for all

operators, and the set of closed terms modulo bisimulation turns into an algebra for the signature of ACP.

We quote from [3, 4] the result that the axiomatisation of Table 1 is sound and complete for the algebra of closed terms modulo bisimulation. Of course, in order to prove this, we have to update all the results that go into this proof in the present setting, and some things do become more difficult (for instance, termination of the associated term rewrite system).

$$a \xrightarrow{a} \surd$$

$$\frac{x \xrightarrow{a} x'}{x + y \xrightarrow{a} x'} \qquad \frac{y \xrightarrow{a} y'}{x + y \xrightarrow{a} y'} \qquad \frac{x \xrightarrow{a} \surd}{x + y \xrightarrow{a} \surd} \qquad \frac{y \xrightarrow{a} \surd}{x + y \xrightarrow{a} \surd}$$

$$\frac{x \xrightarrow{a} x'}{x \cdot y \xrightarrow{a} x' \cdot y} \qquad \frac{x \xrightarrow{a} \surd}{x \cdot y \xrightarrow{a} y}$$

$$\frac{x \xrightarrow{a} x'}{x \parallel y \xrightarrow{a} x' \parallel \Pi^+(y)} \qquad \frac{y \xrightarrow{a} y'}{x \parallel y \xrightarrow{a} \Pi^+(x) \parallel y'} \qquad \frac{x \xrightarrow{a} \surd}{x \parallel y \xrightarrow{a} \Pi^+(y)} \qquad \frac{y \xrightarrow{a} \surd}{x \parallel y \xrightarrow{a} \Pi^+(x)}$$

$$\frac{x \xrightarrow{a} x'}{x \between y \xrightarrow{a} x' \parallel \Pi^+(y)} \qquad \frac{x \xrightarrow{a} \surd}{x \between y \xrightarrow{a} \Pi^+(y)}$$

$$\frac{x \xrightarrow{a} x',\ y \xrightarrow{b} y'}{x \parallel y \xrightarrow{a \mid b} x' \parallel y'} \qquad \frac{x \xrightarrow{a} \surd,\ y \xrightarrow{b} y'}{x \parallel y \xrightarrow{a \mid b} y'} \qquad \frac{x \xrightarrow{a} x',\ y \xrightarrow{b} \surd}{x \parallel y \xrightarrow{a \mid b} x'} \qquad \frac{x \xrightarrow{a} \surd,\ y \xrightarrow{b} \surd}{x \parallel y \xrightarrow{a \mid b} \surd}$$

$$\frac{x \xrightarrow{a} x',\ y \xrightarrow{b} y'}{x \mid y \xrightarrow{a \mid b} x' \parallel y'} \qquad \frac{x \xrightarrow{a} \surd,\ y \xrightarrow{b} y'}{x \mid y \xrightarrow{a \mid b} y'} \qquad \frac{x \xrightarrow{a} x',\ y \xrightarrow{b} \surd}{x \mid y \xrightarrow{a \mid b} x'} \qquad \frac{x \xrightarrow{a} \surd,\ y \xrightarrow{b} \surd}{x \mid y \xrightarrow{a \mid b} \surd}$$

$$\frac{x \xrightarrow{a} x',\ a \notin H}{\partial_H(x) \xrightarrow{a} \partial_H(x')} \qquad \frac{x \xrightarrow{a} \surd,\ a \notin H}{\partial_H(x) \xrightarrow{a} \surd}$$

Table 3. Deduction rules for ACP with pointers ($a, b \in A, a \mid b \in A$).

We can also consider other models of the theory of ACP with pointers, such as a projective limit model. As of yet, we have not considered any model involving internal actions (τ) or empty process (ϵ).

The extension with iteration or more general forms of recursion is straightforward. A preliminary observation that can be made is that the pointer mechanism adds expressive power. For instance, consider the process $a^*\delta$ that keeps on executing action a (the solution of recursive equation $X = a \cdot X$). The process $a^*\delta \parallel b(\pi 1)$ has infinitely many different states (even in the absence of communication), so the set of regular processes is not closed under parallel composition.

Now it is time to look more closely at the mechanism of communication. In particular, we need to look at the mechanism that is called *scope extrusion* in π-calculus. First of all, for the remainder of this article we suppose we have standard read/send communication, so the γ function is only defined on label

pairs (r, s) (read, send) and given by $\gamma(r, s) = \gamma(s, r) = c$ (c for communicate). These labels come with two parameters, a channel name and a data element. If $i \in C$ is a name in a set of channel names, and $d \in D$ is a data element in a data set, we write $r_i(d), s_i(d), c_i(d)$ instead of $r(i, d), s(i, d), c(i, d)$. The way we now model value passing is as follows: suppose a sender process S wants to send a particular value d_0 along channel 1, and the receiver process R wants to receive any value, and process it further. We put

$$S = s_1(d_0) \cdot S' \qquad R = \sum_{d \in D} r_1(d) \cdot R(d).$$

In order to enforce communication, we encapsulate the actions in the set $H = \{r, s\}$. Using the axioms of ACP, we obtain

$$\partial_H(S \parallel R) = c_1(d_0) \cdot \partial_H(S' \parallel R(d_0)),$$

and the value is passed from left to right. If we have a pointer instead of the channel name, it works the same way.

We see that in the receiver process the value d is actually a bound variable, that gets replaced by the particular value d_0 in communication. Now we consider the case where not a value but a pointer is passed, so we have $S = s_1(\pi k) \cdot S'$. The receiver receives this pointer, and then must update its pointers with the correct number. The way we implement this is to have a special pointer $\pi*$ for a receive action. We extend the communication merge to also allow a communication

$$s_1(\pi k) \mid r_1(\pi*) = c_1(\pi k)$$

and then need to increase all pointers in the receive process that point to the receive action by k. The previous example now goes as follows.

$$S = s_1(\pi k) \cdot S' \qquad R = r_1(\pi*) \cdot R'$$

$$\partial_H(S \parallel R) = c_1(\pi k) \cdot \partial_H(S' \parallel \Pi_1^{+k}(R'))$$

Here, the pointer update function Π_1^{+k} increments pointer 1 in the initial position by exactly k, and leaves other pointers unchanged. The equations of this update function are shown in Table 4. But we also need to change the axioms concerning communication, as we have introduced a new form of communication.

So far, we have only defined communication on a matching pair of actions, as in

$$\alpha(\vec{d}, \vec{\pi}) \mid \beta(\vec{d}, \vec{\pi}) = \zeta(\vec{d}, \vec{\pi})$$

where $\gamma(\alpha, \beta) = \zeta$. For the new cases, it is enough to just consider send/receive pairs, as we show in Table 5 below. We add an extra parameter $\pi*$, and add the axioms in this table to the ones of Table 1. The axioms CF2, CM5, CM7 of Table 1 only apply in case there is no parameter $\pi*$ present.

We employ two prefix operators that allow us to introduce symbolic labels to denote communication links. These prefix (binding) operators are as introduced in [1]. The first prefix operator, called *initialisation prefix* is denoted n/v; for

$\Pi_n^{+k}(\pi n) = \pi n + k$	PU1
$\Pi_n^{+k}(\pi m) = \pi m$ if $m \neq n$	PU2
$\Pi_n^{+k}(\alpha(\vec{d}, \vec{\pi})) = \alpha(\vec{d}, \Pi_n^{+\vec{k}}(\pi))$	PU3
$\Pi_n^{+k}(a \cdot x) = \Pi_n^{+k}(a) \cdot \Pi_{n+1}^{+k}(x)$	PU4
$\Pi_n^{+k}(x + y) = \Pi_n^{+k}(x) + \Pi_n^{+k}(y)$	PU5
$\Pi_n^v(v) = \pi n$	PV1
$\Pi_n^v(d) = d$ if $d \neq v$	PV2
$\Pi_n^v(\alpha(\vec{d}, \vec{\pi})) = \alpha(\Pi_n^{\vec{v}}(d), \vec{\pi})$	PV3
$\Pi_n^v(a \cdot x) = \Pi_n^v(a) \cdot \Pi_{n+1}^v(x)$	PV4
$\Pi_n^v(x + y) = \Pi_n^v(x) + \Pi_n^v(y)$	PV5

Table 4. Axioms of pointer update operators ($\alpha \in L \cup \{\delta\}, a \in A$).

$s_i(\vec{d}, \pi k) \mid r_i(\vec{d}, \pi *) = c_i(\vec{d}, \pi k)$	CFπ1
$a \mid b = \delta$ otherwise	CFπ2
$s_i(\vec{d}, \pi k) \mid (r_i(\vec{d}, \pi *) \cdot x) = c_i(\vec{d}, \pi k) \cdot \Pi_1^{+k}(x)$	CM5π1
$a \mid (b \cdot x) = (a \mid b) \cdot x$ otherwise	CM5π2
$(s_i(\vec{d}, \pi k) \cdot x) \mid (r_i(\vec{d}, \pi *) \cdot y) = c_i(\vec{d}, \pi k) \cdot (x \parallel \Pi_1^{+k}(y))$	CM7π1
$(a \cdot x) \mid (b \cdot y) = (a \mid b) \cdot (x \parallel y)$ otherwise	CM7π2

Table 5. Axioms for pointer communication (i a pointer or data value, $\vec{d}$ a data vector, $a, b \in A$

all actions n. It has the effect that $n/v;P$ means to perform action n followed by process P such that inside P the symbolic label v refers to the unique point in time at which the action n happened. This prefix operator can be used to translate the 'new v' prefix of the π-calculus. The second prefix operator is $\mathsf{er}_i^\pi(v)$, called *input prefix* for 'early read', which has the effect of binding variable v to a value that is input. The following rules allow us to eliminate the prefix operators:

$$n/v;P = n \cdot \Pi_1^v(P)$$
$$\mathsf{er}_i^\pi(v);P = r_i(\pi*) \cdot \Pi_1^v(P)$$

We use the operator Π_n^v, which has the effect of replacing a symbolic label (v) by concrete labels (relative pointers) such as πn, $\pi n+1$ etc., depending on the precise depth of the symbolic label inside the process tree. The laws for this operator are shown in Table 4. As for the pointer update operator, we can prove by means of structural induction for all closed terms over the theory of ACP with pointers the following identity:

$$\Pi_n^v(x \parallel y) = \Pi_n^v(x) \parallel \Pi_n^v(y).$$

As we did in [1], we can generalize these prefix operators to the case where more than one initialisation or more than one input takes place. Then we get parallel initialisation or parallel input that is defined (in case of two items) as follows:

$$(n/v \parallel n/w);P = n \cdot n \cdot \Pi_2^v(\Pi_1^w(P)) + n \cdot n \cdot \Pi_1^v(\Pi_2^w(P))$$
$$(\mathsf{er}_i^\pi(v) \parallel \mathsf{er}_j^\pi(w));P = r_i(\pi*) \cdot r_j(\pi*) \cdot \Pi_2^v(\Pi_1^w(P)) + r_j(\pi*) \cdot r_i(\pi*) \cdot \Pi_1^v(\Pi_2^w(P))$$

With the help of these prefix operators, we can now address the π-calculus (see [16, 15]). The main prefix operators of the π-calculus can be defined as follows:

$$\nu v.P \stackrel{\text{def}}{=} \nu/v;P$$
$$\overline{v}(w).P \stackrel{\text{def}}{=} s_v(w) \cdot P$$
$$v(w).P \stackrel{\text{def}}{=} \mathsf{er}_v^\pi(w);P$$

Note that in the middle definition, w is not bound in P. Thus, we obtain the π-calculus by forgetting about the pointer update operators (in the sense of [1]), using these definitions and next, adding τ and τ-laws. We can say that the π-calculus forgets the counting aspect of the present theory and maintains names modulo α-conversion. Notice that an important difference with the π-calculus is that there, in case there is more than one 'new' prefix, they commute, whereas they do not in general, in our case (a more faithful translation is where in such case, we use the parallel initialisation prefix). On the other hand, we can use strong bisimulation as our notion of equivalence. The exact relation will require more research.

The definition of *durational actions* can now be easily added just like we did in [2]:

$$\hat{t} \stackrel{\text{def}}{=} t(\pi 0) \cdot t(\pi 1).$$

3 Client Server System

We describe a simple client-server system. There are two processes: a client and a server. The architecture of the client-server system is shown in Figure 1.

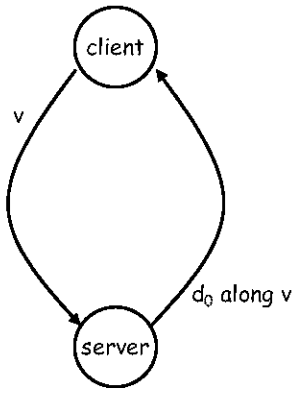

Fig. 1. Client-server system

The idea is that the server holds a certain data value $d_0 \in D$ which is to be sent to the client, but first the client has to inform the server of the communication link to be used. In other words, the client C takes the initiative to establish a communication link with the server S. Once the communication link (v) is established, the client uses this link to listen for a data value to be received from the server.

$$\begin{aligned} C &= n/v; (s(v) \cdot \sum_{d \in D} r_v(d) \cdot C_d) \\ S_d &= \mathsf{er}^{\pi}(w); (s_w(d) \cdot S) \\ CS &= \partial_H(C \parallel S_{d_0}) \end{aligned}$$

where again $H = \{s, r\}$. We assume that the remainders C_d, S do not contain any variables, so that $\Pi_n^v(C_d) = C_d$ and $\Pi_m^w(S) = S$. We proceed to calculate

the system CS.

$$\begin{aligned}
CS &= \partial_H(C \parallel S_{d_0}) \\
&= \partial_H(n \cdot s(\pi 1) \cdot \sum_{d \in D} r_{\pi 2}(d) \cdot C_d \parallel r(\pi *) \cdot s_{\pi 1}(d_0) \cdot S) \\
&= n \cdot \partial_H(s(\pi 1) \cdot \sum_{d \in D} r_{\pi 2}(d) \cdot C_d \parallel r(\pi *) \cdot s_{\pi 1}(d_0) \cdot S) \\
&= n \cdot c(\pi 1) \cdot \partial_H(\sum_{d \in D} r_{\pi 2}(d) \cdot C_d \parallel s_{\pi 2}(d_0) \cdot S) \\
&= n \cdot c(\pi 1) \cdot c_{\pi 2}(d_0) \cdot \partial_H(C_{d_0} \parallel S)
\end{aligned}$$

We see the system behaves as expected. Abstracting from pointers, i.e. applying the operator $\phi_{\{c\}}$ yields

$$\phi_{\{c\}}(CS) = n \cdot c() \cdot c(d_0) \cdot \phi_{\{c\}}(\partial_H(C_{d_0} \parallel S))$$

The above example can be used to illustrate the working of the pointer mechanism of ACP with pointers, as shown in Figure 2. The process terms are represented as boxes, composed by sequential composition (indicated by lines with a dot, going from left to right) and parallel composition (indicated by $\parallel$). The thin lines show how each occurrence of a label, for example represented by $\pi 1$ or $\pi 2$, points backwards to an action such as n or $r(\pi *)$ that marks the defining occurrence of a variable. The pointer tells how many dots have to be skipped: for example $\pi 1$ means to skip one dot, $\pi 2$ means to skip two dots, and so on. The analogy with De Bruijn sequences is obvious. Figure 2 shows the second transition of the above calculation. The figure also explains what happens when a $s(\pi 1)$ action and an $r(\pi *)$ action are combined into a single $c(\pi 1)$ action: inside the subterm that sequentially follows the $r(\pi *)$ action, all pointers that were pointing to the $r(\pi *)$ are updated in order to make them point to the marked position communicated by the send action. If a pointer is at a distance of L_2 from the receive action $r(\pi *)$ and if the send action is at a distance of L_1 from the defining occurence of the label then the updated pointer points to the action that is located $L_1 + L_2$ steps away.

4 Client broker server system

We describe a slightly more complicated system which contains three processes: a client, a broker and a server. The architecture of this system is shown in Figure 3.

Initially the server holds a certain data value $d_0 \in D$ which must be sent to the client. In this example, it is assumed that initially the client does not know the server and conversely, the server does not know the client. The broker B is known to all parties, however, by means of fixed communication links 1,2. The server S takes the initiative by sending its own communication link w to the broker and similarly the client C sends its communication link v to the

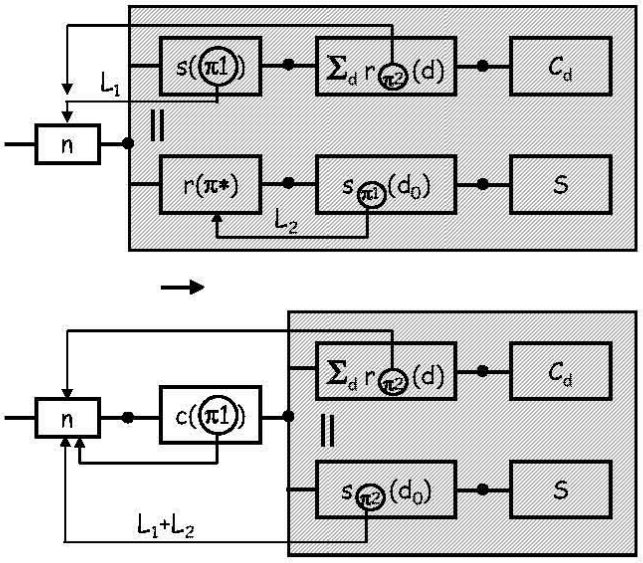

Fig. 2. The pointer mechanism in action.

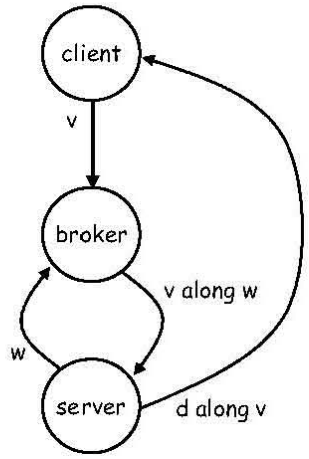

Fig. 3. Client broker server system.

broker. After having received both links, the broker informs the server about the communication link v of the client (the broker can do this since it knows w now). Now the broker has performed its task and returns to the idle state. Finally the

server sends the data value d_0 to the client along the link v.

$$\begin{aligned} C &= n_c/v; (s_2(v) \cdot \sum_{d \in D} r_v(d)) \\ B &= (\mathsf{er}_1^\pi(w) \parallel \mathsf{er}_2^\pi(v)); s_w(v) \\ S_d &= n_s/w; (s_1(w) \cdot \mathsf{er}_w^\pi(v); s_v(d) \\ CBS &= \partial_H(C \parallel S_{d_0} \parallel B) \end{aligned}$$

where again $H = \{s, r\}$.

Here we use the parallel input prefix as explained above. It means the two inputs can take place in arbitrary order. The translation into ACPπ of these specifications is as follows.

$$\begin{aligned} C &= n_c \cdot s_2(\pi 1) \cdot \sum_{d \in D} r_{\pi 2}(d) \\ B &= r_1(\pi *) \cdot r_2(\pi *) \cdot s_{\pi 2}(\pi 1) + r_2(\pi *) \cdot r_1(\pi *) \cdot s_{\pi 1}(\pi 2) \\ S_d &= n_s \cdot s_1(\pi 1) \cdot r_{\pi 2}(\pi *) \cdot s_{\pi 1}(d) \end{aligned}$$

It is possible to calculate the specification of system *CBS*. We show this in the calculations to follow.

$$\begin{aligned} CBS = n_c \cdot \partial_H(s_2(\pi 1) \cdot \sum_{d \in D} r_{\pi 2}(d) \parallel B \parallel S) + \\ + n_s \cdot \partial_H(C \parallel B \parallel s_1(\pi 1) \cdot r_{\pi 2}(\pi *) \cdot s_{\pi 1}(d_0)) \end{aligned}$$

$$
\begin{aligned}
&= n_c \cdot \Big(n_s \cdot \partial_H(s_2(\pi 2) \cdot \sum_{d\in D} r_{\pi 3}(d) \parallel B \parallel s_1(\pi 1) \cdot r_{\pi 2}(\pi *) \cdot s_{\pi 1}(d_0)) \\
&\qquad + c_2(\pi 1) \cdot \partial_H(\sum_{d\in D} r_{\pi 2}(d) \parallel r_1(\pi *) \cdot s_{\pi 1}(\pi 3) \parallel S) \Big) \\
&+ n_s \cdot \Big(n_c \cdot \partial_H(s_2(\pi 1) \cdot \sum_{d\in D} r_{\pi 2}(d) \parallel B \parallel s_1(\pi 2) \cdot r_{\pi 3}(\pi *) \cdot s_{\pi 1}(d_0)) \\
&\qquad + c_1(\pi 1) \cdot \partial_H(C \parallel r_2(\pi *) \cdot s_{\pi 3}(\pi 1) \parallel r_{\pi 2}(\pi *) \cdot s_{\pi 1}(d_0)) \Big) \\
&= n_c \cdot \Big(n_s \cdot \Big(c_2(\pi 2) \cdot \partial_H(\sum_{d\in D} r_{\pi 3}(d) \parallel r_1(\pi *) \cdot s_{\pi 1}(\pi 4) \parallel s_1(\pi 2) \cdot r_{\pi 3}(\pi *) \cdot s_{\pi 1}(d_0)) \\
&\qquad + c_1(\pi 1) \cdot \partial_H(s_2(\pi 3) \cdot \sum_{d\in D} r_{\pi 4}(d) \parallel r_2(\pi *) \cdot s_{\pi 3}(\pi 1) \parallel r_{\pi 2}(\pi *) \cdot s_{\pi 1}(d_0)) \Big) \\
&\qquad + c_2(\pi 1) \cdot n_s \cdot \partial_H(\sum_{d\in D} r_{\pi 3}(d) \parallel r_1(\pi *) \cdot s_{\pi 1}(\pi 4) \parallel s_1(\pi 1) \cdot r_{\pi 2}(\pi *) \cdot s_{\pi 1}(d_0)) \Big) \\
&+ n_s \cdot \Big(n_c \cdot \Big(c_2(\pi 1) \cdot \partial_H(\sum_{d\in D} r_{\pi 2}(d) \parallel r_1(\pi *) \cdot s_{\pi 1}(\pi 3) \parallel s_1(\pi 3) \cdot r_{\pi 4}(\pi *) \cdot s_{\pi 1}(d_0)) \\
&\qquad + c_1(\pi 2) \cdot \partial_H(s_2(\pi 2) \cdot \sum_{d\in D} r_{\pi 3}(d) \parallel r_2(\pi *) \cdot s_{\pi 4}(\pi 1) \parallel r_{\pi 3}(\pi *) \cdot s_{\pi 1}(d_0)) \Big) \\
&\qquad + c_1(\pi 1) \cdot n_c \cdot \partial_H(s_2(\pi 1) \cdot \sum_{d\in D} r_{\pi 2}(d) \parallel r_2(\pi *) \cdot s_{\pi 4}(\pi 1) \parallel r_{\pi 3}(\pi *) \cdot s_{\pi 1}(d_0)) \Big) \\
&= n_c \cdot \Big(n_s \cdot \Big(c_2(\pi 2) \cdot c_1(\pi 2) \cdot \partial_H(\sum_{d\in D} r_{\pi 4}(d) \parallel s_{\pi 3}(\pi 4) \parallel r_{\pi 3}(\pi *) \cdot s_{\pi 1}(d_0)) \\
&\qquad + c_1(\pi 1) \cdot c_2(\pi 3) \cdot \partial_H(\sum_{d\in D} r_{\pi 4}(d) \parallel s_{\pi 3}(\pi 4) \parallel r_{\pi 3}(\pi *) \cdot s_{\pi 1}(d_0)) \Big) \\
&\qquad + c_2(\pi 1) \cdot n_s \cdot c_1(\pi 1) \cdot \partial_H(\sum_{d\in D} r_{\pi 4}(d) \parallel s_{\pi 2}(\pi 4) \parallel r_{\pi 2}(\pi *) \cdot s_{\pi 1}(d_0)) \Big) \\
&+ n_s \cdot \Big(n_c \cdot \Big(c_2(\pi 1) \cdot c_1(\pi 3) \cdot \partial_H(\sum_{d\in D} r_{\pi 3}(d) \parallel s_{\pi 4}(\pi 3) \parallel r_{\pi 4}(\pi *) \cdot s_{\pi 1}(d_0)) \\
&\qquad + c_1(\pi 2) \cdot c_2(\pi 2) \cdot \partial_H(\sum_{d\in D} r_{\pi 3}(d) \parallel s_{\pi 4}(\pi 3) \parallel r_{\pi 4}(\pi *) \cdot s_{\pi 1}(d_0)) \Big) \\
&\qquad + c_1(\pi 1) \cdot n_c \cdot c_2(\pi 1) \cdot \partial_H(\sum_{d\in D} r_{\pi 2}(d) \parallel s_{\pi 4}(\pi 2) \parallel r_{\pi 4}(\pi *) \cdot s_{\pi 1}(d_0)) \Big)
\end{aligned}
$$

$$
\begin{aligned}
&= n_c \cdot \Big(n_s \cdot (c_2(\pi 2) \cdot c_1(\pi 2) + c_1(\pi 1) \cdot c_2(\pi 3)) \\
&\qquad \cdot c_{\pi 3}(\pi 4) \cdot \partial_H(\sum_{d \in D} r_{\pi 5}(d) \parallel s_{\pi 5}(d_0)) \\
&\qquad + c_2(\pi 1) \cdot n_s \cdot c_1(\pi 1) \cdot c_{\pi 2}(\pi 4) \cdot \partial_H(\sum_{d \in D} r_{\pi 5}(d) \parallel s_{\pi 5}(d_0)) \Big) \\
&+ n_s \cdot \Big(n_c \cdot (c_2(\pi 1) \cdot c_1(\pi 3) + c_1(\pi 2) \cdot c_2(\pi 2)) \\
&\qquad \cdot c_{\pi 4}(\pi 3) \cdot \partial_H(\sum_{d \in D} r_{\pi 4}(d) \parallel s_{\pi 4}(d_0)) \\
&\qquad + c_1(\pi 1) \cdot n_c \cdot c_2(\pi 1) \cdot c_{\pi 4}(\pi 2) \cdot \partial_H(\sum_{d \in D} r_{\pi 3}(d) \parallel s_{\pi 3}(d_0)) \Big) \\
&= n_c \cdot (n_s \cdot (c_2(\pi 2) \cdot c_1(\pi 2) + c_1(\pi 1) \cdot c_2(\pi 3)) \cdot c_{\pi 3}(\pi 4) \cdot c_{\pi 5}(d_0) \\
&\qquad + c_2(\pi 1) \cdot n_s \cdot c_1(\pi 1) \cdot c_{\pi 2}(\pi 4) \cdot c_{\pi 5}(d_0)) \\
&+ n_s \cdot (n_c \cdot (c_2(\pi 1) \cdot c_1(\pi 3) + c_1(\pi 2) \cdot c_2(\pi 2)) \cdot c_{\pi 4}(\pi 3) \cdot c_{\pi 4}(d_0) \\
&\qquad + c_1(\pi 1) \cdot n_c \cdot c_2(\pi 1) \cdot c_{\pi 4}(\pi 2) \cdot c_{\pi 3}(d_0))
\end{aligned}
$$

We show one trace of the system in Figure 4, and the complete transition system, with pointers visualized in Figure 5.

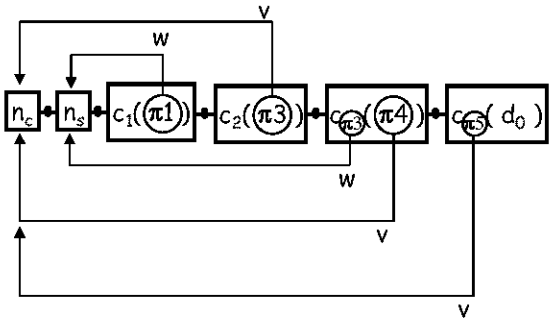

Fig. 4. One trace of the CBS system

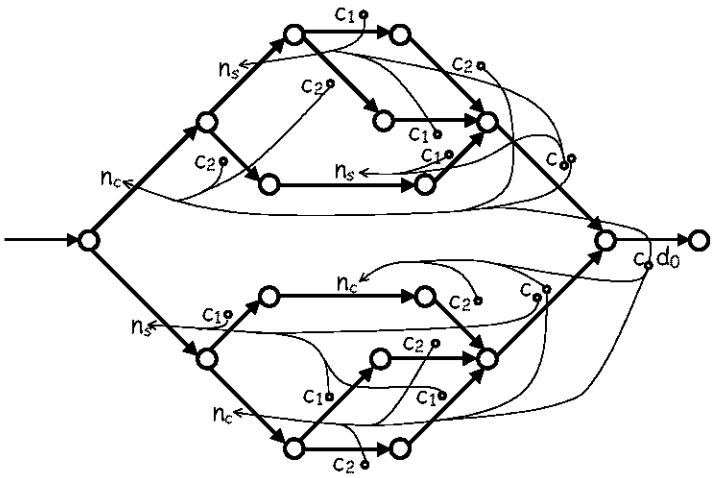

Fig. 5. Transition system of the CBS system

5 Mobile phone system

We model Milner's example (Sect. 8.2 in [15]: Mobile phones, pp.80-83). The architecture of the mobile phone system is shown in Figure 6. There are four processes: a car, a central controller and two transmitters.

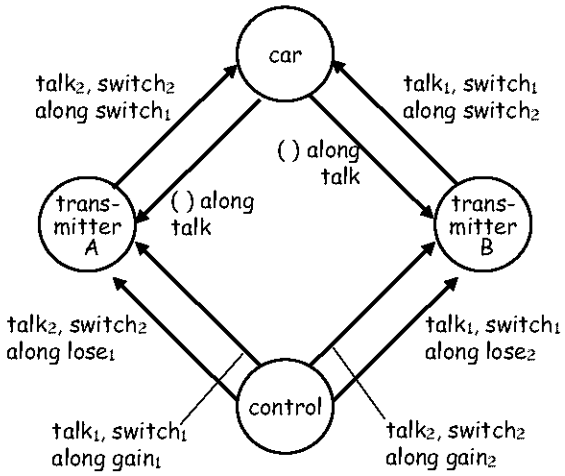

Fig. 6. Mobile phone system

There are several communication links, identified by variables talk_1, gain_1, switch_1, lose_1, talk_2, gain_2, switch_2, and lose_2. The idea is that the transmitter Trans is a kind of base station which knows the comunication links 'talk' and 'switch' that can be used to communicate with the car. The transmitter also knows the ports 'gain' and 'lose' to be used for communicating with the central controller, called Control1 (if it is in its first state) or Control2 (otherwise). The transmitter receives contentless messages from the car via 'talk'. If the transmitter receives via 'lose' a command from the central controller, it informs (via 'switch') the car of the new 'talk' and 'switch' ports (t and s) to be used. After this, the transmitter goes into the idle state. In the idle state, if the transmitter receives the 'gain' command from the central controller, this means that the transmitter is informed of the new 'talk' and 'switch' ports to be used next.

First we give the defining equations of the car and the transmitter. Process names are parameterised: Car has two parameters, Trans has four parameters and Idtrans (the transmitter in its idle state) has two parameters.

$$\begin{aligned}
\mathrm{Car}_{\mathrm{talk},\mathrm{switch}} &= s_{\mathrm{talk}}(d_0) \cdot \mathrm{Car}_{\mathrm{talk},\mathrm{switch}} \\
&\quad + \mathsf{er}^{\pi}_{\mathrm{switch}}(t); (\mathsf{er}^{\pi}_{\mathrm{switch}}(s); \mathrm{Car}_{t,s}) \\
\mathrm{Idtrans}_{\mathrm{gain},\mathrm{lose}} &= \mathsf{er}^{\pi}_{\mathrm{gain}}(t); (\mathsf{er}^{\pi}_{\mathrm{gain}}(s); \mathrm{Trans}_{t,s,\mathrm{gain},\mathrm{lose}}) \\
\mathrm{Trans}_{\mathrm{talk},\mathrm{switch},\mathrm{gain},\mathrm{lose}} &= r_{\mathrm{talk}}(d_0) \cdot \mathrm{Trans}_{\mathrm{talk},\mathrm{switch},\mathrm{gain},\mathrm{lose}} \\
&\quad + \mathsf{er}^{\pi}_{\mathrm{lose}}(t); (\mathsf{er}^{\pi}_{\mathrm{lose}}(s); (s_{\mathrm{switch}}(t).s_{\mathrm{switch}}(s).\mathrm{Idtrans}_{\mathrm{gain},\mathrm{lose}}))
\end{aligned}$$

Our equations are almost a direct translation of Milner's example, except for the fact that we do not convey two or more communication links in a single step. So instead of, for example, $\mathsf{er}^{\pi}_{\mathrm{switch}}(t,s);(\ldots)$ we use two steps, writing $\mathsf{er}^{\pi}_{\mathrm{switch}}(t);(\mathsf{er}^{\pi}_{\mathrm{switch}}(s);(\ldots))$.

Next we give the remaining equations. Car1 and Car2 represent the two distinct states of the car. Similarly Control1 and Control2 represent the two distinct states of the controller.

$$
\begin{aligned}
\mathrm{Car1} &= \mathrm{Car}_{\mathrm{talk}_1,\mathrm{switch}_1}\\
\mathrm{Car2} &= \mathrm{Car}_{\mathrm{talk}_2,\mathrm{switch}_2}\\
\mathrm{TransA} &= \mathrm{Trans}_{\mathrm{talk}_1,\mathrm{switch}_1,\mathrm{gain}_1,\mathrm{lose}_1}\\
\mathrm{TransB} &= \mathrm{Trans}_{\mathrm{talk}_2,\mathrm{switch}_2,\mathrm{gain}_2,\mathrm{lose}_2}\\
\mathrm{IdtransA} &= \mathrm{Idtrans}_{\mathrm{gain}_1,\mathrm{lose}_1}\\
\mathrm{IdtransB} &= \mathrm{Idtrans}_{\mathrm{gain}_2,\mathrm{lose}_2}\\
\mathrm{Control1} &= s_{\mathrm{lose}_1}(\mathrm{talk}_2)\cdot s_{\mathrm{lose}_1}(\mathrm{switch}_2)\cdot\\
&\qquad s_{\mathrm{gain}_2}(\mathrm{talk}_2)\cdot s_{\mathrm{gain}_2}(\mathrm{switch}_2)\cdot\mathrm{Control2}\\
\mathrm{Control2} &= s_{\mathrm{lose}_2}(\mathrm{talk}_1)\cdot s_{\mathrm{lose}_2}(\mathrm{switch}_1)\cdot\\
&\qquad s_{\mathrm{gain}_1}(\mathrm{talk}_1)\cdot s_{\mathrm{gain}_1}(\mathrm{switch}_1)\cdot\mathrm{Control1}\\
\mathrm{MPS1} &= \partial_H((n/\mathrm{talk}_1 \parallel n/\mathrm{switch}_1 \parallel n/\mathrm{gain}_1 \parallel n/\mathrm{lose}_1 \parallel\\
&\qquad n/\mathrm{talk}_2 \parallel n/\mathrm{switch}_2 \parallel n/\mathrm{gain}_2 \parallel n/\mathrm{lose}_2);\\
&\qquad (\mathrm{Car1} \parallel \mathrm{TransA} \parallel \mathrm{IdtransB} \parallel \mathrm{Control1}))
\end{aligned}
$$

Where H is defined in the usual way. We use here the parallel initialisation prefix, denoting that these initialisations can take place in arbitrary order. In the following calculations, we fix one such order, given by the textual order.

We use some obvious shorthands, such as $\vec{n}/\mathrm{talk}_1 \mathinner{..} \mathrm{lose}_2$, which is an abbreviation for $n/\mathrm{talk}_1 \parallel n/\mathrm{switch}_1 \parallel n/\mathrm{gain}_1 \parallel n/\mathrm{lose}_1 \parallel n/\mathrm{talk}_2 \parallel n/\mathrm{switch}_2 \parallel n/\mathrm{gain}_2 \parallel n/\mathrm{lose}_2$. Using this we can perform a number of rewritings, one for each name introduction.

$$
\begin{aligned}
\mathrm{MPS1} &= \partial_H(\,\vec{n}/\mathrm{talk}_1 \mathinner{..} \mathrm{lose}_2;\ \left(\begin{array}{l}
\mathrm{Car}_{\mathrm{talk}_1,\mathrm{switch}_1}\\
\parallel \mathrm{Trans}_{\mathrm{talk}_1,\mathrm{switch}_1,\mathrm{gain}_1,\mathrm{lose}_1}\\
\parallel \mathrm{Idtrans}_{\mathrm{gain}_2,\mathrm{lose}_2}\\
\parallel s_{\mathrm{lose}_1}(\mathrm{talk}_2).s_{\mathrm{lose}_1}(\mathrm{switch}_2)\\
\quad .\ s_{\mathrm{gain}_2}(\mathrm{talk}_2).s_{\mathrm{gain}_2}(\mathrm{switch}_2)\\
\qquad .\ \mathrm{Control2}
\end{array}\right)\,)\\
&= \underbrace{n\cdot\cdots\cdot n}_{8\text{ times}}\cdot\partial_H\left(\begin{array}{l}
\mathrm{Car}_{\pi 8,\pi 7}\\
\parallel \mathrm{Trans}_{\pi 8,\pi 7,\pi 6,\pi 5}\\
\parallel \mathrm{Idtrans}_{\pi 2,\pi 1}\\
\parallel s_{\pi 5}(\pi 4)\cdot s_{\pi 6}(\pi 4)\cdot s_{\pi 4}(\pi 6)\cdot s_{\pi 5}(\pi 6)\\
\quad \cdot \Pi_{12\ \ldots 5}^{\mathrm{talk}_1\mathinner{..}\mathrm{lose}_2}(\mathrm{Control2})
\end{array}\right)
\end{aligned}
$$

where $\Pi_{12\ \ldots 5}^{\mathrm{talk}_1\mathinner{..}\mathrm{lose}_2}(\mathrm{Control2})$ abbreviates $\Pi_{12}^{\mathrm{talk}_1}(\,\Pi_{11}^{\mathrm{switch}_1}(\,\Pi_{10}^{\mathrm{gain}_1}(\,\Pi_{9}^{\mathrm{lose}_1}(\,\Pi_{8}^{\mathrm{talk}_2}(\,\Pi_{7}^{\mathrm{switch}_2}(\,\Pi_{6}^{\mathrm{gain}_2}(\,\Pi_{5}^{\mathrm{lose}_2}(\mathrm{Control2}))))))))$. The effect is that symbolic labels are replaced by pointers, which (for the top-level of the $\parallel$ construct) means that the substitutions are given by the following table:

talk_1	$\pi 8$
switch_1	$\pi 7$
gain_1	$\pi 6$
lose_1	$\pi 5$
talk_2	$\pi 4$
switch_2	$\pi 3$
gain_2	$\pi 2$
lose_2	$\pi 1$

At this point, a real communication step can be performed. Basically, there is the following behaviour: any number of talk communications can be executed, and at each point in this sequence, a hand-over procedure can be started. We will write out part of the system, where the handover procedure starts before the first talk action.

$$\begin{aligned}\text{MPS1} &= n^8 \cdot \partial_H \left(\begin{array}{l} \text{Car}_{\pi 8,\pi 7} \\ \| \ \text{Trans}_{\pi 8,\pi 7,\pi 6,\pi 5} \\ \| \ \text{Idtrans}_{\pi 2,\pi 1} \\ \| \ s_{\pi 5}(\pi 4) \cdot s_{\pi 6}(\pi 4) \cdot s_{\pi 4}(\pi 6).s_{\pi 5}(\pi 6) \\ \quad \cdot \Pi_{12 \ .. \ 5}^{\text{talk}_1 \ .. \ \text{lose}_2}(\text{Control2}) \end{array} \right) \\ &= n^8 \cdot \Bigg(c_{\pi 8}(d_0) \cdot \ldots + \\ &\qquad c_{\pi 5}(\pi 4) \cdot \partial_H \left(\begin{array}{l} \text{Car}_{\pi 9,\pi 8} \\ \| \ r_{\pi 6}(\pi *) \cdot s_{\pi 9}(\pi 6) \cdot s_{\pi 10}(\pi 2) \cdot \text{Idtrans}_{\pi 10,\pi 9} \\ \| \ \text{Idtrans}_{\pi 3,\pi 2} \\ \| \ s_{\pi 6}(\pi 4) \cdot s_{\pi 4}(\pi 6) \cdot s_{\pi 5}(\pi 6) \cdot \Pi_{12 \ .. \ 5}^{\text{talk}_1 \ .. \ \text{lose}_2}(\text{Control2}) \end{array} \right) \Bigg)\end{aligned}$$

Following the talk communication $c_{\pi 8}(d_0)$, another one is possible, then denoted as $c_{\pi 9}(d_0)$, or, alternatively, the handover procedure is started, then denoted by $c_{\pi 6}(\pi 5)$. Here, we show the state of the system after immediate handover initiation, by $c_{\pi 5}(\pi 4)$.

The entire sequence of events during handover is shown in Figure 7. Our calculations have reached the point immediately before the first message that is labeled $c_{\text{lose}_1}(\text{talk}_2)$.

After the handover procedure, a new series of talk communications can be started, but alternatively, immediately a new handover procedure can be started. We show a message sequence chart showing this in Figure 8.

We do not show the complete calculations here, but instead, show the transition system of the complete system. In Figure 9, the two states that are filled in solid correspond, with all 1-indexed actions replaced by 2-indexed actions and vice versa: in the uppermost state, talk_1-actions can be started, in the other one, talk_2-actions. Similarly, the two states marked with a cross are related, and the two states with a thicker circle. Thus, the complete behaviour of the system is exhibited.

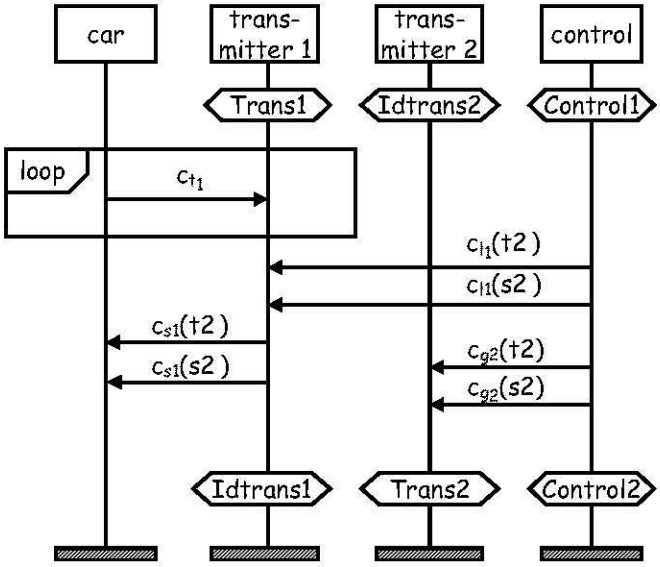

Fig. 7. Sequence of events during hand-over

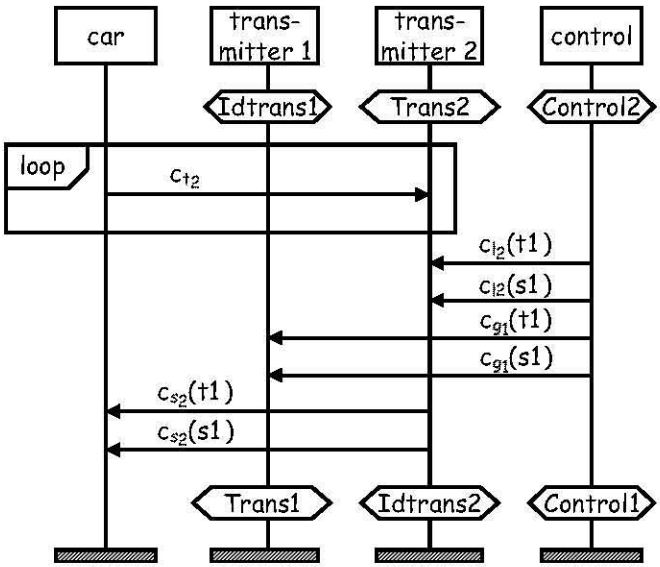

Fig. 8. Sequence of events during hand-over (continued)

6 Conclusion

We have seen we have introduced a process algebra that can handle aspects of mobile processes and mobile communication links. Secondly, as we expected,

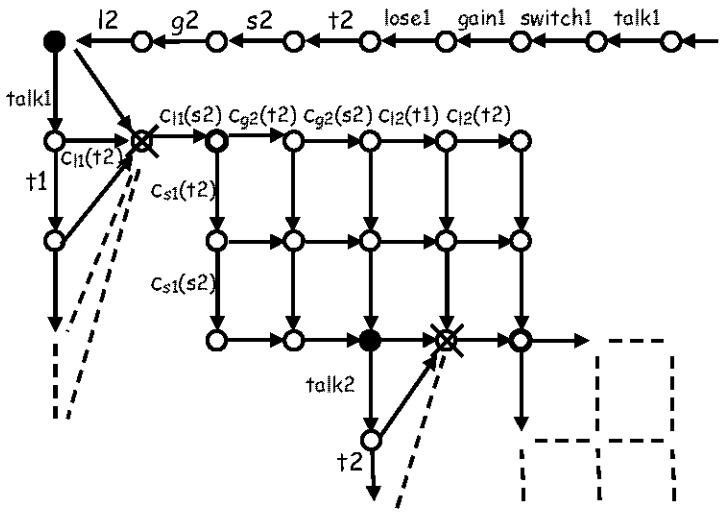

Fig. 9. Transition system of mobile phone system

the formalism is operating on a level which is better suited for tool supported calculations that for manual calculations. This article can further understanding of calculi for mobility, and can further implementations and further work concerning such calculi.

References

1. J.C.M. Baeten and J.A. Bergstra. On sequential composition, action prefixes and process prefix. *Formal Aspects of Computing*, 6(3):250–268, 1994.
2. J.C.M. Baeten and J.A. Bergstra. Deadlock behaviour in split and ST bisimulation semantics. In I. Castellani and C. Palamidessi, editors, *Proceedings EXPRESS'98*, number 16 in Electronic Notes in Theoretical Computer Science, pages 101–114, Nice, 1998. Elsevier. http://www.elsevier.nl/locate/entcs/volume16.2.html.
3. J.C.M. Baeten and C. Verhoef. Concrete process algebra. In S. Abramsky, D.M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 4, pages 149–269. Oxford University Press, 1995.
4. J.C.M. Baeten and W.P. Weijland. *Process Algebra.* Number 18 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1990.
5. H.P. Barendregt. *The Lambda Calculus (Its Syntax and Semantics).* Number 103 in Studies in Logic and the Foundations of Mathematics. North-Holland, 1984.
6. H.P. Barendregt. Lambda calculi with types. In S. Abramsky, D.M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, pages 117–309. Oxford University Press, 1992.
7. J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Information and Control*, 60(1/3):109–137, 1984.

8. J.A. Bergstra and A. Ponse. Grid protocol specifications. In B. Möller and J.V. Tucker, editors, *Prospects for Hardware Foundations*, number 1546 in Lecture Notes in Computer Science, pages 278–308. Springer Verlag, 1998.
9. N.G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae*, 34(5):381–392, 1972.
10. N. Busi, R.J. van Glabbeek, and R. Gorrieri. Axiomatising ST-bisimulation semantics. In E.-R. Olderog, editor, *Proceedings of the IFIP TC2 Working Conference on Programming Concepts, Methods and Calculi (PROCOMET'94)*, number 56 in IFIP Transactions A, pages 169–188. North-Holland, Amsterdam, 1994.
11. Ph. Darondeau and P. Degano. Causal trees. In G. Ausiello, M. Dezani-Ciancaglini, and S. Ronchi Della Rocca, editors, *Proceedings ICALP'89*, number 372 in Lecture Notes in Computer Science, pages 234–248. Springer Verlag, 1989.
12. F. Kamareddine and R.P. Nederpelt. On stepwise explicit substitution. *Foundations of Computer Science*, 4:197–240, 1993.
13. F. Kamareddine and R.P. Nederpelt. A useful λ-notation. *Theoretical Computer Science*, 155:85–109, 1996.
14. J. McKinna and R. Pollack. Pure types systems formalized. In M. Bezem and J.F. Groote, editors, *Proceedings TLCA'93*, number 664 in LNCS, pages 289–305. Springer Verlag, 1993.
15. R. Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.
16. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Information and Computation*, 100:1–77, 1992.
17. R.P. Nederpelt, J.H. Geuvers, and R.C. de Vrijer. *Selected Papers on Automath*. North-Holland, 1994.
18. A.M. Pitts. A fresh approach to representing syntax with static binders in functional programming. In *Proceedings ICFP'2001*. ACM Press, 2001.
19. R. Pollack. Closure under alpha-conversion. In *Proceedings TYPES'93*, number 806 in LNCS. Springer Verlag, 1993.
20. B. Victor. *A Verification Tool for the Polyadic π-Calculus*. Licentiate thesis, Department of Computer Systems, Uppsala University, Sweden, 1994. Available as report DoCS 94/50.