A Calculus of Communicating Systems with Label Passing—Ten Years After

Uffe H. Engberg Mogens Nielsen

BRICS∗
Department of Computer Science†
University of Aarhus
Ny Munkegade
DK-8000 Aarhus C, Denmark

Abstract

This note is dedicated to Robin Milner. We take the opportunity of looking back on a ten year old report of ours on an extension of CCS with label passing.

1 Preface

This note is dedicated to Robin Milner on his 60th birthday. On this occasion we have taken the opportunity of commenting on a report of ours from 1986, on an extension of Milner’s CCS with label passing [EN86].

The challenging problem of extending CCS with a notion of label (or channel) passing was motivated and formulated by Milner himself already in 1979, before CCS was published. At the time Milner was spending his sabbatical at the University of Aarhus, and in discussions with one of us (Nielsen) we looked upon various ideas for such extensions of CCS, but did not manage to get any of these to work properly. In 1985 Engberg was looking for a subject for his MSc thesis, and we decided to make another attempt at the six year old problem and ideas. At that time, a few attempts had already been made towards calculi with notions of mobility (as it was later called by Milner et al. in [MPW89]), notably the parametric channels by Astesiano and Zucca [AZ84] and the LNET formalism by Kennaway and

∗Basic Research in Computer Science, Centre of the Danish National Research Foundation.
†e-mail address: {engberg,mnielsen}@brics.dk, fax: ++45 89 42 32 55
Sleep [KS83], which was a kind of hybrid between CCS and the actor model of Hewitt. However, no attempt had been made towards a calculus with a proper algebraic theory, and following the ideas of Milner, we took it as our main challenge to develop such a calculus, which would enhance the expressibility of CCS by introducing a notion of channel passing, and at the same time preserve as much as possible of the algebraic theory of CCS. As we put it at the time: “With all the thought behind the elegance of CCS this is a sound principle to apply in any attempt to extend CCS”.

So our goals were reasonably concrete. The theory of CCS was there, and we only designed our own small set of examples, which we used as “expressiveness benchmarks” (Eratosthenes’ sieve, a pushdown store, and an extension of Milner’s translation of an imperative programming language into CCS, [Mil80]). We tried out several approaches but most of them failed in one or more respects, and we never managed to get quite the calculus we were after. However, a particularly promising approach was presented and studied in the MSc thesis of Engberg finished at the turn of the year 1985–86. Engberg’s thesis introduced an extension of CCS, ECCS, with an operational semantics, some suggestions for behavioural equivalences, and our benchmark examples of modelling and reasoning using the calculus. The essential contents of the thesis was reported only in a very preliminary form in an Technical Report from University of Aarhus, [EN86], which was accompanied in Engberg’s MSc thesis by 112 pages of handwritten proofs. We never produced a version of the report suitable for publication.

The technical contributions of ECCS are not of much interest today ten years after. It has long since been surpassed by the π-calculus, as introduced in the seminal paper of Milner, Parrow and Walker [MPW89], and developed by Milner himself and many other researchers. However, we thought it might be of some interest to look back on our report as part of the early developments of calculi of mobile processes.

In their seminal paper on the π-calculus [MPW89], Milner et al. comment in their conclusion:

Engberg and Nielsen (1986) did not publish their report, and it has not received due attention, probably because its treatment of constants, variables, and names is somewhat difficult. Many features of the π-calculus are due to them, in particular the replacement of the CCS relabelling by syntactic substitution (crucial for the formulation of the semantic rules); the semantic treatment of scope extrusion; the extension of the definition of bisimulation to account for name parameters; the definition of
strong bisimilarity (which they call simply ‘strong equivalence’); and the soundness of most algebraic laws. We made many failed attempts to depart from their formulation. Our contribution has been to remove all discrimination among constant names, variable names, and values, yielding a more basic calculus; to discriminate between ground and non-ground equivalence (needed to replace the constant-variable discrimination); to strengthen the algebraic laws—in particular the expansion law—in order to achieve complete equational theories; to encode value-computation in the calculus in a tractable way (with the help of a new match construct); and to provide rather simple encodings of functional calculi—the λ-calculus and combinatory algebra.

One of our aims is to point out that the gap between [EN86] and the π-calculus was much bigger than indicated by the quote above. Milner et al. actually not only provided a much simpler and more elegant calculus with a proper theory, they also corrected a number of conceptual and technical mistakes of ours, as will be pointed out in this note.

Also, we would like to recall some of the few ideas from [EN86] which seem to have survived in the development of calculi with mobility, probably those summarised by our goal “to extend CCS to allow passing of individual channels, viewing restriction as a formal binder, and to allow dynamic change of scope of such binders in connection with communication”.

Our modest aim is to give a few comments on the background and the development of these early calculi of mobile processes seen from our own perspective. We have no ambition of covering the wide range of further developments of Milner and many other researchers over the past ten years. Victor and Nestmann maintain a Web based bibliography of works on the π-calculus [VN].

We have chosen to start by quoting the entire Introduction from [EN86]. The reason is not that the text is particularly interesting reading today from a technical point of view, but we think it gives an account of the kind of reasoning we went through at the time, and hence some of the motivation for ECCS. Next, we present some of the contents of [EN86], and elaborate on the similarities and differences between ECCS and the π-calculus with respect to syntax, derivation rules, behaviour equivalence, and expressiveness. And we finish off by quoting and commenting on the entire Conclusion from [EN86].
2 Introduction from 1986

In the original version of CCS, as presented in [Mil80], *structured* dynamically evolving configurations such as the pushdown store can be obtained by means of recursion and the chaining combinator.

It is less clear that the same can be obtained for *unstructured* dynamically evolving configurations like the example studied in [Mil80, chapter 9], translating a concurrent programming language with unboundedly many concurrent activations of a single procedure into CCS. There it was pointed out that a solution would be to allow the passage of communication links as values between one agent and another, but that CCS was probably defective in this respect. It was also noted that the usefulness of such a solution was not limited to language translations.

In [DG83] it is mentioned that in general, the exchange of ports (communication links) between agents, would be a natural way to model the exchange of communication capabilities.

In later versions of CCS (see [Mil83, Mil84]) a more basic calculus, which allows infinite summation but not direct value communication, was introduced. It was shown how the original version—a richer calculus—could be encoded. Value communication and manipulation was encoded essentially by indexing the labels and the agent identifiers. Labels or communication links could also be encoded (as special cases of values). A similar approach has been made by [AZ84] which conceptually only differs a little from Milner’s approach. A very different approach called LNET is presented in [KS83]. LNET might be described as a hybrid of actor languages and CCS.

We have made a different approach for several reasons.

Although the later basic calculus in a sense allows the passage of communication links as values between one agent and another, Milner himself notices in [Mil84]: “It is quite certain that the slender syntax of our basic calculus and even the derived notations which we have considered, are not sufficient always to present such applications [with passage of communication links] in a lucid way”. Most of the problems are left for the “programmer”.

Our approach will be more in keeping with the original version of CCS, at the same time widening the connection to the lambda-calculus and reducing the number of primitive operators (no relabelling) without loss of expressiveness. In [Mil80] Milner asks the question whether CCS’s primitive constructs are the smallest possible set and says that they need a re-examination. Since we have not got the relabelling operator we to some extent deal with this question. It is our belief that the parts of our approach which concern this could be done for the basic calculus too.
We will now discuss what requirements the new calculus allowing passage of communication links should meet. It will be referred to as ECCS (Extended CCS).

In what follows there is a slight syntactical difference to CCS. $\alpha ? x . B$ is written for $\alpha x . B$, where $x$ is bound by $?$ and its scope is $B$ meaning that a value can be received at $\alpha$. Similarly we write $\alpha ! v . B$ for sending a value.

To get a first idea of what we mean by allowing passage of communication links (labels) consider the example:

$$B_0 \parallel B_1 \equiv \alpha ! \lambda \beta ? x . \delta ! x . \text{nil} \parallel \alpha ? y . \beta ! y . \text{nil}$$

which is a CCS program. According to CCS it can develop like this:

$$B_0 \parallel B_1 \downarrow \tau$$
$$\beta ? x . \delta ! x . \text{nil} \parallel \beta ! \lambda \cdot \text{nil}$$
$$\downarrow \tau$$
$$\delta ! \lambda . \text{nil} \parallel \text{nil}$$

If we replace $\lambda$ by $\lambda$ it is no longer a CCS program, but we wish such communications of labels to be possible in ECCS. If $x$ and $y$ are replaced by $x$ and $y$—variables qualifying over labels—we expect the program to be able to develop in the same way:

$$\alpha ! \lambda . \beta ? x . \delta ! x . \text{nil} \parallel \alpha ? y . \beta ! y . \text{nil}$$
$$\downarrow \tau$$
$$\beta ? x . \delta ! x . \text{nil} \parallel \beta ! \lambda . \text{nil}$$
$$\downarrow \tau$$
$$\delta ! \lambda . \text{nil} \parallel \text{nil}$$

*Communication of labels* would be of no use if it was not possible to use a received label for later communication, so the following modification of the example,

$$B_0 \parallel B_1 \equiv \alpha ! \lambda . \lambda ? x . \delta ! x . \text{nil} \parallel \alpha ? y . y ! \lambda . \text{nil}$$

should be possible so that a development could be:

$$B_0 \parallel B_1 \downarrow \tau$$
$$\lambda ? x . \delta ! x . \text{nil} \parallel \lambda ! \lambda . \text{nil}$$
$$\downarrow \tau$$
$$\delta ! \lambda . \text{nil} \parallel \text{nil}$$
Up till now there has probably not been any problem in understanding these basic requirements. This is due to the simplicity of the examples. In [DG83], the following more complicated example is studied:

\[(B_0 \mid B_1) \setminus \alpha \mid B_2 \equiv (\lambda?y.y!\cdot\text{nil} \mid \lambda!\alpha.\alpha?.\text{nil}) \setminus \alpha \mid \lambda!\alpha.\alpha?.\text{nil}\]

The agent \(B_0\) can receive a label at \(\lambda\) and the label is bound to the variable \(y\). If the received label \(\alpha\) comes from the agent \(B_1\) it agrees with our intuition if the system upon the communication results in:

\[(\alpha!.\text{nil} \mid \alpha?.\text{nil}) \setminus \alpha \mid \lambda!\alpha.\alpha?.\text{nil}.\]

But what should the result look like if the label received originates from the outside agent \(B_2\)? Should it be possible to pass \(\alpha\) from \(B_2\) to \(B_0\)? If the system instead looked like

\[(B_0 \mid B_3) \setminus \beta \mid B_2 \equiv (\lambda?y.y!\cdot\text{nil} \mid \lambda!\beta.\beta?.\text{nil}) \setminus \beta \mid \lambda!\alpha.\alpha?.\text{nil}, \alpha \neq \beta\]

it would seem natural if the result was \((\alpha!.\text{nil} \mid B_3) \setminus \beta \mid \alpha?.\text{nil}\). The labels visible for agent \(B_2\) are the same in both cases. From the viewpoint of \(B_2\) there seems no reason why \((B_0 \mid B_1) \setminus \alpha\) and \((B_0 \mid B_3) \setminus \beta\) should behave differently. It will therefore be a central requirement to ECCS, that the name of a label restricted should be of no importance to the behaviour in the same way as change of bound variable in \(\lambda?x.\) — does not influence the behaviour in CCS. This also seems natural if one takes up the attitude that one is communicating via links and that the communications via a certain link should be the same no matter what name is chosen for that link. In terms of experiments on machines as sketched in [Mil80]: the buttons are the same no matter what names are printed on them.

The same question arises in a different situation and the problems seem closely connected. Let \(\lambda?y.B \equiv \lambda?y.(\alpha!5.\text{nil} \mid \alpha?.x.y!\cdot\text{nil}) \setminus \alpha\). What should the result look like after a label (say \(\alpha\)) is received at \(\lambda\) and substituted for \(y\) in \(B\)? The situation is very similar to the one above, except that dependence of the names of the restricted labels is displaced to the substitution. We therefore demand the same independence of actual names used for restriction when substituting a label.

We will now study one further requirement to ECCS through an example mentioned in [Mil84]. Consider the agent \(A\) managing some resources \(R_i\) \((1 \leq i \leq n)\) which signal to \(A\) via \(\lambda\) when they are available. Other agents make requests for resources to \(A\) via \(\gamma\). Let \(Q\) be such a potentially requesting agent. The situation can be pictured as:
where the resource $R_i$ is accessed through $\alpha_i$. A common solution is to write the system as

$$(Q \mid A \mid R_1 \mid \ldots \mid R_n) \setminus A,$$

where $A = \{\alpha_i; 1 \leq i \leq n\}$

and to let $A$ somewhere contain a subexpression like $\gamma!i.B$ meaning that $A$ communicates the index of an available resource via $\gamma$; and let $Q$ contain a subexpression like $\gamma?x.(\ldots \alpha_x!\ldots \alpha_x?\ldots)$ where $\gamma?k$ means receiving the index and using it for communication with resource $R_x$ via $\alpha_x$. The problem of such solutions passing indexes as a kind of identification is that all potential resources must be known at the time when the administrator $A$ and the system is written in order not to mix up indexes. Furthermore the family of indexed labels $\{\alpha_i\}$ must be known in advance. To illustrate this we consider a very simple system with a requesting agent and two resources.

We leave out the details of $A$. $A$ would in this example be $\{\alpha_1, \alpha_2\}$.

If one wants to add a new resource $R_3$ to the system one must inspect the system to see that communications between requesting agents is done via labels of type $\alpha_x$ and that $\alpha_1, \alpha_2$ already is used. It is not enough to know the way they communicate (the communication protocol they use) and that requests for resources are done to $A$ via $\gamma$ and resource availability is reported via $\lambda$. Furthermore $A$ must be extended with $\{\alpha_3\}$ if '3' is used to identify $R_3$.

This shows a certain lack of modularity which we want to avoid.

Therefore it should be possible to write the different agents independently of each other only knowing the interface to $A$, i.e. $R_i$ knows $\lambda$ of $A$ and $Q$
knows \(\gamma\) of \(A\). As a consequence it must be possible for \(R_i\) to have a label which in a certain sense is unique in all contexts and which later can be used as communication link between \(R_i\) and \(Q\). In addition the label shall remain unique or private to \(R_i\) and \(Q\) after it is communicated through \(A\) via \(\lambda\) and \(\gamma\), (except of course if it is communicated further from either \(Q\) or \(R_i\)).

Last but not least we impose the restriction to keep as close as possible to Milner’s CCS—e.g. preserving as many as possible of the algebraic properties of CCS. This requirement is actually quite independent of the extension we seek. With all the thought behind the elegance of CCS this is a sound principle to apply in any attempt to extend CCS.

We will now give an idea of how ECCS can be made in order to meet these requirements.

Milner has already drawn attention to the connection between the binder “?” in \(\alpha?x\) of CCS and the binder “\(\lambda\)” in the lambda-calculus ([Mil80, p.49]). He also introduced a textual substitution postfix which has similar characteristics as the substitution prefix of the lambda-calculus (see [Mil80, p.67]) namely: when applied, change of bound variables is done as necessary to avoid clashes. It is clear that the substitution postfix formally can be handled along the lines of the substitution prefix of the lambda-calculus as long as we are only concerning the binding construct \(\alpha?x\). But it is less clear that the binding construct \(\text{fix } X\) introduced for recursion in [Mil83, Mil84] can be handled formally within the same framework, especially when \(X = \{X_i; i \in I\}\) an \(I\)-indexed family of distinct variables where \(I\) is an uncountable set. One of our aims will therefore be to lift the results for the extension where \(\text{fix}\) appears as binder too (though only for finite \(X\)).

In order to meet the requirement of independence of actual names used for restriction we will furthermore widen the idea of bound and free occurrences to include labels as well, with “\(\\backslash\)” as the binding symbol for labels. In the lambda-calculus a central notion is \(\alpha\)-convertibility between functions with respect to bound variables. The idea is that functions which are equal “up to bound variables” denote the same function when applied to the same arguments. With our requirement that behaviour expressions which are equal “up to bound labels” should behave equally, it seems natural to extend the notion of \(\alpha\)-convertibility to include labels bound by “\(\\backslash\)”.

The close relationship between substitution of variables and \(\alpha\)-convertibility (in the following just convertibility) will therefore also be generalised to substitution of labels. At the same time we thereby obtain the possibility to change unbound labels of a behaviour expression (i.e. the sort) and can therefore omit the postfixed relabelling operator. For the third requirement (possibility to have a unique or private label, to communicate it and for it to
remain unique) notice that the label $\alpha$ in some sense is unique to $B$ in $B \setminus \alpha$ since $\alpha$ can be used only for internal communication and cannot interfere with other $\alpha$’s appearing in any contexts $B \setminus \alpha$ could be in. So in order to communicate $\alpha$ it must be possible to extend $\alpha$’s scope to include the recipient. This and the remaining uniqueness is obtained through a (minor) extension of the inference rules.

To put the comments above differently, we want to extend CCS to allow passing of individual channels, viewing restriction as a formal binder, and to allow dynamic change of scope of such binders in connection with communication.

In order to get an idea of the possibilities of ECCS we turn back to the example of an administrator $A$ and some agents $Q_i$ requesting some resources $R_j$ via the administrator.

We will make the simplifying assumption that it does not matter what resource a requesting agent gets, though the resources may be implemented differently as long as they obey the same communication protocol. If the administrator uses the “first-come-first-served” policy, it can be implemented as a FIFO-queue where a requesting agent enters the queue at the rear and leaves it at the front when a resource is available:

$$A \equiv (\gamma?newreq.T(newreq, \sigma) | \sigma!.nil) \setminus \sigma,$$

$$T \equiv \text{fix } X(\langle \text{oldreq,x} \rangle (\langle \gamma?newreq.X(newreq, \sigma) | \langle x?..\alpha?x.\alpha!e.X(\ldots) \rangle \setminus \alpha \rangle, \alpha \neq \lambda$$

Each slanted name denotes a label variable ($\lambda, \gamma \neq \sigma$).

The administrator can be viewed as consisting of a series of elements each containing a waiting agent (oldreq). At the rear new requests are received ($\gamma?newreq$) and a new element created. The front element receives the name of an available resource ($\lambda?freeres$) and passes it on to the waiting agent (oldreq!freeres). After doing this it signals to the next ($\sigma!$) that it is the new front element now. Notice that the elements at the front and rear can serve the agents and resources concurrently.

We now turn to the parts of the resources and requesting agents which concern the communication between them and the administrator. A resource could look something like:

$$R_j \equiv \text{fix } X\tilde{p}((\lambda!\alpha.\alpha?x.\ldots.\alpha!e.X(\ldots)) \setminus \alpha), \alpha \neq \lambda$$

and a requesting agent:

$$Q_i \equiv \ldots (\gamma!\beta.\beta?x.x!e'.\ldots.x?y.\ldots) \setminus \beta\ldots, \beta \neq \gamma$$
<table>
<thead>
<tr>
<th>Form</th>
<th>$B''$</th>
<th>$\text{FV}(B'')$</th>
<th>$L(B'')$</th>
<th>$B(B'')$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inaction</td>
<td>nil</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>Action</td>
<td>$\lambda y.B$</td>
<td>$\text{FV}(B) - {y}$</td>
<td>$L(B) \cup {\lambda}$</td>
<td>$B(B) \cup {y}$</td>
</tr>
<tr>
<td></td>
<td>$x?y.B$</td>
<td>$(\text{FV}(B) - {y}) \cup {x}$</td>
<td>$L(B) \cup {x}$</td>
<td>$B(B) \cup {x}$</td>
</tr>
<tr>
<td></td>
<td>$\lambda e.B$</td>
<td>$\text{FV}(B) \cup {e}$</td>
<td>$L(B) \cup L(e)$</td>
<td>$B(B) \cup B(e)$</td>
</tr>
<tr>
<td></td>
<td>$x!e.B$</td>
<td>$\text{FV}(B) \cup {e} \cup {x}$</td>
<td>$L(B) \cup L(e)$</td>
<td>$B(B) \cup B(e)$</td>
</tr>
<tr>
<td></td>
<td>$\tau.B$</td>
<td>$\text{FV}(B)$</td>
<td>$L(B) \cup {\tau}$</td>
<td>$B(B)$</td>
</tr>
<tr>
<td>Summation</td>
<td>$B + B'$</td>
<td>$\text{FV}(B) \cup \text{FV}(B')$</td>
<td>$L(B) \cup L(B') \cup B(B) \cup B(B')$</td>
<td>$B(B) \cup B(B')$</td>
</tr>
<tr>
<td>Composition</td>
<td>$B</td>
<td>B'$</td>
<td>$\text{FV}(B) \cup \text{FV}(B')$</td>
<td>$L(B) \cup L(B') \cup B(B) \cup B(B')$</td>
</tr>
<tr>
<td>Restriction</td>
<td>$B \setminus \alpha$</td>
<td>$\text{FV}(B)$</td>
<td>$L(B) - {\alpha}$</td>
<td>$B(B) \cup {\alpha}$</td>
</tr>
<tr>
<td>Recursion</td>
<td>$\text{fix}_X \bar{p}E(\bar{e})$</td>
<td>$(\text{FV}(E) - {X, \bar{p}}) \cup {\bar{e}}$</td>
<td>$L(E) \cup L(\bar{e})$</td>
<td>$B(E) \cup {X, \bar{p}} \cup B(\bar{e})$</td>
</tr>
<tr>
<td>Conditional</td>
<td>$\text{if } f \text{ then } B$</td>
<td>${f} \cup \text{FV}(B) \cup \text{FV}(B')$</td>
<td>$L(B) \cup L(B') \cup B(f) \cup B(B) \cup B(B')$</td>
<td>$B(B) \cup B(B')$</td>
</tr>
</tbody>
</table>

Table 1: Syntax table for ECCS behaviour expressions.

The underlined actions correspond to the communication protocol between requesting agents and resources for this special example. The other shown actions concern the communication with the administrator. The label $\alpha$ in the resource is restricted and therefore unique or private for $R_j$. Upon sending it to the administrator, its scope is extended to include the administrator (in accordance with $\text{Com} \rightarrow (3)$ and $\text{Res} \rightarrow (2)$ in section 3.3). A new name is possibly chosen in order not to interfere with other labels within the new scope such that it remains private to the resource. Afterwards it is passed on to the requesting agent at the front of the queue and the scope includes the agent in the same manner as before. The same happens when the requesting agent sends its private label—which acts like an identification—to the administrator. Notice that, after it has served an agent, a resource restores itself such that it sends a new private label when reporting that it is available.

3 ECCS and the $\pi$-calculus

In the following we would like to present our original calculus ECCS, and to elaborate a little on its relationship to the $\pi$-calculus of Milner et al. [MPW89]. Let us start by the syntax.
\begin{center}
\begin{tabular}{|c|c|c|c|}
\hline
\alpha & Kind & fn(\alpha) & bn(\alpha) \\
\hline
\tau & Silent & \emptyset & \emptyset \\
\bar{x} y & Free Output & \{x, y\} & \emptyset \\
x(y) & Input & \{x\} & \{y\} \\
\bar{x}(y) & Bound Output & \{x\} & \{y\} \\
\hline
\end{tabular}
\end{center}

Table 2: The $\pi$-calculus actions.

3.1 Syntax

We chose to base our calculus on three types of expressions: behaviour expressions, label expressions and ordinary data expressions. Accordingly, we assumed the following disjoint sets of variables and labels: label variables (ranged over by $x, y, \ldots$), process variables ($X, Y, \ldots$) and labels ($\alpha, \beta, \ldots$).

In order to present the syntax and semantics of ECCS, we introduced the following rather clumsy conventions of associating symbols with domains of values and variables:

- $a$: label or label variable,
- $b, c$: variable or label (bindable elements),
- $e$: label, label variable (label expressions) or a data expression built from constant and function symbols as usual (e.g. used as actual parameters to recursive process definitions),
- $f$: data expression alone,
- $p, x, y$: label variables or data variables (e.g. used for formal parameters to recursive process definitions),
- $v$: label or data,
- $M, N$: label, label variable, data expression, process variable or a indexed recursive process definition,

$B, E$: behaviour expressions.

Based on this, the complete syntax of ECCS is listed in Table 1. As you see, our syntax followed closely that of CCS from [Mil80]. Besides minor points, like our use of the $? \text{ and } !$ notation for input and output actions respectively, the main difference is, of course, the introduction of label variables on top of the other data variables from CCS, and the correspondingly more
complex definitions of free variables, FV, free labels (sort), L, and bound variables and labels, B. Table 1 was based on conventions and assumptions like the following.

i) \( \{e\} \) denotes the set of (free) variables occurring in expression \( e \).

ii) Vectors of variables are allowed in recursive process definitions (only!), where \( \langle X_1, \ldots, X_n \rangle \) is abbreviated \( \tilde{X} \).

iii) Labels and label variables are also allowed as actual parameters to a recursion expression.

iv) Two elements of the syntax are said to be comparable if they are the same type like e.g. a label \( \alpha \) and a label variable \( x \), a behaviour expression \( E \) and a process variable \( X \), etc.

v) Formal parameters, \( \tilde{p} \), and actual parameters, \( \tilde{e} \), of a recursion definition are tacitly assumed to be of the same dimension and corresponding elements to be comparable.

vi) In a conditional expression no label or label variable may be contained in \( f \) to form a boolean expression.

For comparison we recapitulate the syntax of the \( \pi \)-calculus of Milner et al. Agents or process expressions of the \( \pi \)-calculus are defined as follows:

\[
P ::= 0 \mid x(y).P \mid \tilde{x}y.P \mid \tau.P \mid P + Q \mid P \mid Q \mid (x)P \mid [x = y]P \mid A(y_1, \ldots, y_n),
\]

where \( w, x, y, z \) range over (link) names, \( P \) and \( Q \) range over agents and \( A \) over agent identifiers. Here \( 0 \) represents inaction. The action prefixes \( x(y). \), \( \tilde{x}y \), and \( \tau \) corresponds to input-, (free) output- and silent-prefix respectively. The restriction of \( x \) to \( P \) is denoted \( (x)P \) to stress that it is regarded as a name-binding operator. \( [x = y]P \) represents matching of names and behaves like \( P \) if \( x \) equals \( y \) and otherwise like \( 0 \).

Each agent identifier \( A \) must be associated a unique defining equation \( A(y_1, \ldots, y_n) \overset{\text{def}}{=} P \) with \( \text{fn}(P) \subseteq \{y_1, \ldots, y_n\} \), where Milner et al. use \( \text{fn}(P) \) to denote the free names of \( P \). \( \text{fn}(P) \) is defined structurally on \( P \) similarly as FV for ECCS in Table 1. In the same way \( \text{bn}(P) \) denotes the bound names of \( P \) and \( \text{n}(P) \) is just the names of \( P \) whether free or bound. The free and bound names of the action prefixes can be seen in the first three lines of Table 2.
The simplifications of Milner et al. are particularly striking on the syntactical level. They removed non-label expressions, blurred the distinction between labels and label variables, and replaced the fix recursion construct with defining equations. Consequently the number of binders and syntactic categories was diminished considerably and a much more elegant language was obtained.

The only possible label expressions of ECCS were labels and label variables. In contrast Milner et al. allowed a matching construct (which in our setup would correspond to testing labels for equality), and used it to give tractable encodings of data computations and a powerful form of expansion law.

Following [Mil80] we did not state any language of data expressions (non-label expressions) in the presentation of the calculus, and as such the calculus can be regarded as parametrised in the language of data expressions. However, given the data encodings of Milner et al. and successors, it should be clear that ECCS can be simulated in the \( \pi \)-calculus.

Though we deliberately chose the fix construct to handle recursion we could have used a system of defining equations. For a defining equation \( A(\tilde{p}) \overset{\text{def}}{=} E \) we would as Milner et al. have to demand \( \text{FV}(E) \subseteq \{\tilde{p}\} \). However, we would also have to demand \( E \) to have no free labels, i.e. \( L(E) = \emptyset \), since different occurrences of \( A \) can be in the scope of restrictions and so possibly subject to different \( \alpha \)-conversions. This somewhat unpleasant condition is avoided with the fix recursion construct. In follow up papers [Mil90, Mil91], Milner obtained an even more basic calculus using the replication operator “!”.

### 3.2 Substitution, Conversion

As indicated earlier we relied heavily on notions of substitution and convertability. Not only were these notions from the \( \lambda \)-calculus generalised to behaviour expressions regarding input and fix as variable binders, but more importantly restriction was viewed as a proper (label) binder as in \( \lambda \)-calculus. This crucial idea was suggested to us by Matthew Hennessy in discussions on the development of ECCS.

In ECCS \([M/b]B\) is defined provided \( M \) is comparable to \( b \) and denotes the result of syntactically substituting \( M \) for all (free) occurrences of the bindable element \( b \) in the expression \( B \), with change of comparable bound elements as necessary to avoid clashes. The substitution prefix is defined not only for behaviour expressions but also for atomic elements such as labels, data-values and variables.
Having generalised substitution, we generalised $\alpha$-conversion to handle the three forms of binders, and defined a convertibility relation, $\text{cnv}$, over behaviour expressions accordingly. Substitution and conversion were shown to satisfy the expected properties such as preservation of convertability under substitution.

In the $\pi$-calculus a substitution is simply a function $\sigma$ from names to names which is the identity except on a finite set of names. $P\sigma$ denotes the simultaneous substitution of $z\sigma$ for $z$ in $P$ with change of bound names to avoid captures. Milner et al. write $\{y/x\}$ for $\sigma$ in the special case where $\sigma$ is $y$ on $x$ and the identity elsewhere. $\equiv_\alpha$ denotes the $\alpha$-convertibility relation on agents. Also these $\pi$-calculus notions of substitution and $\alpha$-conversion were shown by Milner et al. to satisfy the expected properties.

However, even if essentially the same kinds of results were obtained for the ECCS and the $\pi$-calculus notions of substitution and $\alpha$-conversion, the removal of the fix recursion construct, non-label expressions and the distinction between labels and label variables in the $\pi$-calculus, simplified the formal treatment of these notions considerably in the $\pi$-calculus.

### 3.3 Derivations

The operational rules of ECCS are shown in Table 3. Before relating them to those of the $\pi$-calculus a few comments are needed.

First of all, we were looking for a definition of an action relation $B \Gamma \rightarrow B'$ over programs, i.e. for $B$ and $B'$ such that $\text{FV}(B) = \text{FV}(B') = \emptyset$, much in the spirit of CCS, where for a program $B$, the resulting behaviour expression $B'$ is also a program. This is not the case with ECCS, which we considered a drawback! So, we thought of $B \overset{\lambda v}{\rightarrow} B'$ as “$B$ becomes $[v/x]B'$ under $\lambda?v$ where $v$ is any value $x$ can assume, i.e. $v$ is a value comparable to $x$”, and our definition of behavioural equivalence was deliberately defined only in terms of derivation rules $B \overset{\Gamma}{\rightarrow} B'$ for which $B$ is a program—see later.

When looking at the axioms and inference rules in Table 3 defining the atomic action relation please recall that we only write $[M/b]$ when $M$ and $b$ are comparable. So for instance the inference rules $\text{Com} \rightarrow (2)$ are only defined when $v$ and $x$ are comparable. Similarly the axiom $\text{Act} \rightarrow (1)$ is only defined for $y$ and $x$ comparable.

Finally, let us quote one technical proposition and four of our original notes on the action relation:

**Proposition 1** If $B \overset{\Gamma}{\rightarrow} B'$ is a part of an inference which ensures $C \overset{\Gamma}{\rightarrow} C'$ then $\text{FV}(C) = \emptyset$ implies $\text{FV}(B) = \emptyset$. 
i) By Proposition 1 it is clear that if Act→(1) is the basis of an inference which ensures an action of a program then FV(λ?x.B) = ∅ and an arbitrarily chosen y will therefore be just as good as x. Furthermore it will not interfere with variables in another part of the program since, also by Proposition 1, any other part of the program which can form an action cannot have free variables. For instance if D = C | λ?x.B is a program and D λ?x C | B' then x cannot be free in C. Therefore it is ensured in Com→(2) that v is substituted in the “right” place. Com→(1), Com→(2) correspond to those of CCS.

ii) The reason for letting (λ?x.B, [y/x]B) be in the relation λ?y is that we wish α-convertible programs to be behaviourally equivalent. This could have been obtained through a modification of the definition of strong equivalence, but we have found it more convenient here.

iii) Two inference rules are added to CCS: Com→(3) and Res→(2). They make it possible to extend the scope of a label. Res→(2) cancels the restriction and for the reasons mentioned in i) it does not matter which variable is chosen (as long as it is a label variable of course). In Com→(3) the restriction is reintroduced and the label made available to the recipient. The actual name of the label is chosen such that it does not interfere with other names in the new scope by the condition α ∉ L(Bi). Notice that the original name can be chosen if it does not appear in the environment outside the old and inside the new scope.

iv) The definition depends heavily on the substitution prefix and its properties as seen in Act→(1), Com→(2), Com→(3) and Res→(2). Most of all it depends on the property that a label or variable which “passes through” a bound occurrence by substitution changes the names of the bound occurrence and “passes” on, thereby avoiding any conflict.

So much for the derivation rules of ECCS. The corresponding rules of the π-calculus are shown in Table 4. Comparing the transition rules of Table 3 and Table 4 the interesting parts are the treatments of “scope extension”, as we called it, or “scope extrusion” as it was later called by Milner et al.

Notice the close correspondence between the action symbols, Γ, used in the action relation and the prefix constructs of ECCS, also following the spirit of CCS. Not having the ECCS possibility of distinguishing between λv! and λv→ in defining the transition relations of the π-calculus, Milner et al. introduced a fourth kind of action, bounded output $\bar{x}(y)$, which has
Ina → nil has no atomic actions

Act → (1) \( \lambda x.B \xrightarrow{\lambda x?} [y/x]B \)
(2) \( \lambda !v.B \xrightarrow{\lambda !v!} B \)
(3) \( \tau.B \xrightarrow{\tau} B \)

Sum → L:
\[
\frac{B_0 \xrightarrow{\Gamma} B'_0}{B_0 + B_1 \xrightarrow{\Gamma} B'_0}
\]

Com → (1)L:
\[
\frac{B_0 \xrightarrow{\Gamma} B'_0}{B_0 \parallel B_1 \xrightarrow{\Gamma} B'_0 \parallel B_1}
\]
(2)L:
\[
\frac{B_0 \xrightarrow{\lambda \alpha} B'_0, B_1 \xrightarrow{\lambda \alpha} B'_1, \alpha \notin L(B_i)}{B_0 \parallel B_1 \xrightarrow{\alpha / x} (B'_0 \parallel B'_1) \setminus \alpha}
\]

Res → (1)
\[
\frac{B \xrightarrow{\Gamma} B', \alpha \notin L(\Gamma)}{B \setminus \alpha \xrightarrow{\Gamma} B \setminus \alpha}
\]
(2)
\[
\frac{B \xrightarrow{\lambda \alpha} B', \lambda \neq \alpha}{B \setminus \alpha \xrightarrow{\lambda \alpha} [\alpha / \alpha]B'}
\]

Rec → [fix ˜X ˜p ˜E/ ˜X][˜v/ ˜p]E \xrightarrow{\Gamma} B

Con → L:
\[
\frac{B_0 \xrightarrow{\Gamma} B'_0}{\text{if true then } B_0 \text{ else } B_1 \xrightarrow{\Gamma} B'_0}
\]

Table 3: The ECCS transition rules. Each L-rule has a symmetric R-rule.

no corresponding prefix form in the syntax of the calculus, but captures the idea that a private name \( y \) is output on the port \( \bar{x} \). With this notation the OPEN and CLOSE rules of Table 4 correspond closely to the Res→(2) and Com→(2) rules of Table 3, essentially using different syntax for making the same semantic distinction between “normal” output and output carrying a restricted name out of its scope. The actions of the \( \pi \)-calculus are summarised in Table 2.

Also notice that both sets of transition rules follow the late instantiation scheme, i.e. instantiation (by a label or data-value) of a variable bound by an input prefix is deferred to the time of inferring internal communication (Com→(2) and Com→(3)). At first glance CLOSE appears to be relatively different from Com→(3) and less late. However, Milner et al. actually define the transition relations using the following rule:

\[
\frac{P \xrightarrow{\bar{x}(w)} P' \quad Q \xrightarrow{\bar{x}(z)} Q'}{P \parallel Q \xrightarrow{\bar{x}(w)} (P' \parallel Q' \{z/w\})}
\]  
\( w = z \) or \( w \notin fn(Q') \).
Table 4: The $\pi$-calculus Rules of Actions. Rules involving the binary operators $+$ and $|$ additionally have symmetric forms.

Due to the elegance of their formulation of the system of transition rules, they are able to replace the rule with CLOSE. The issue of early versus late instantiation was introduced in [MPW89, MPW91].

Also, many of our concerns on interference of variables were avoided in the $\pi$-calculus by dropping the distinction between labels and variables and by introducing the explicit side condition in PAR rule of Table 4, ensuring that free variables are not accidentally bound in the COM or CLOSE rule. Since we worked on programs (with no free variables) we did not need a similar condition in Com$\rightarrow$(1) as explained in note i) to the action relation.

The following theorem from [EN86] establishes an important connection between substitution, conversion and the action relations in ECCS:

**Theorem 2**

(a) $C \xrightarrow{\Gamma} C' \Rightarrow [\delta/\gamma]C \xrightarrow{[\delta/\gamma]\Gamma} C'' \xrightarrow{\text{cnv}} [\delta/\gamma]C''$
(b) The following diagram commutes in the sense that if $B_0 \xrightarrow{\Gamma} B_1$, then $B_0'$ exists such that $B_0 \xrightarrow{\Gamma} B_0' \xrightarrow{\Gamma} B_1'$ and vice versa:

\[
\begin{array}{c}
B_0 \xrightarrow{\Gamma} B_1 \\
\downarrow \Gamma & \downarrow \Gamma \\
B_0' \xrightarrow{\Gamma} B_1'
\end{array}
\]

(c) $[\delta/\gamma]C \xrightarrow{\Gamma} D$ implies that there exist $\Gamma'$ and $D'$ such that $C \xrightarrow{\Gamma'} D'$ and $[\delta/\gamma]D' \xrightarrow{\Gamma'} \Gamma' = \Gamma$.

As later pointed out to us by David Walker, there was a mistake in our formulation of Theorem 2 (c) which is not correct as stated in the case where $\Gamma = \tau$. As a counterexample consider

$[\delta/\gamma](\gamma ? | \delta!) \xrightarrow{\tau} \text{nil}$.

However, the statement holds provided $\delta$ is not in the sort of $C$ as correctly formulated for the $\pi$-calculus in Lemma 4 from Part II [MPW89] where also the rest of Theorem 2 is shown for the $\pi$-calculus.

### 3.4 Behavioural Equivalences

Since our overall goal was a generalisation of the algebraic theory of CCS, our search for an appropriate operational semantics for ECCS was done in parallel with a corresponding search for appropriate generalisations of Milner’s notions of strong and observational equivalences from [Mil80]. However, whereas our operational semantics changed several times, our generalisations of the behavioural equivalences were actually quite stable. The notions we adopted just seemed the most natural generalisations to us, but looking back, it is obvious that there were several alternatives which we did not even think of, some of them already pointed out in [MPW89]. If we focus on the strong equivalence, our generalisation looked as given in Table 5. To get a nicer formulation of the equivalence, the last two lines were originally joined in a single line (relying on a notation for the union of input and output actions). For clarity and the sake of a later comment, it is here presented with two separate cases.

Our definition was accompanied in [EN86] by claims of a number of properties expected for a proper lifting of the algebraic theory of CCS, e.g. the following three theorems.
Theorem 3

(a) \sim is an equivalence relation  
(b) \sim_{k+1} B_1 implies \sim_k B_1  
(c) B_0 \sim B_1 implies \sim_k B_1 for all k  
(d) B_0 \text{cnv} B_1 implies B_0 \sim B_1

Also, we claimed that we almost had a congruence in the following formal sense, with closure under general substitution, Theorem 4, as “perhaps the hardest to accept”:

Theorem 4 \sim B_0 B_1 implies [\delta/\gamma] \sim B_0 [\delta/\gamma] B_1.

Theorem 5 \sim B_0 B_1 implies \lambda x.B_0 \sim \lambda x.B_1 for all x \in \text{labels}.

The corresponding definition of strong ground bisimilarity adopted by Milner et al. is given in Table 6.

Apart from the fact that the two definitions refer to two different syntactic languages, there are a few more interesting differences worth commenting. In retrospect it is clear that our definition of strong equivalence was unsatisfactory for several reasons.

First of all, our definition was presented along the lines of the definition in the original version of CCS, i.e. in terms of a decreasing sequence of equivalence relations, and not in the more elegant Park and Milner style of strong bisimulation [Par81, Mil83]. The reason was not that we were unaware of this style of definition, but our proof skills were simply insufficient to see how to carry through the proofs for a bisimulation version of our definition. What we failed to see was how to define suitable bisimulation relations in the cases involving restriction. Milner et al. elegantly solved this by using inductively defined bisimulation relations essentially allowing proof techniques which we only found possible with the given definition. Their insight also led to the introduction of an auxiliary definition of strong bisimulation up to restriction.

Another reason is a consequence of the previously mentioned problem in Theorem 2. As we shall see shortly, our strong equivalence does not in general preserve substitution of labels and consequently it cannot, with the present formulation, be a congruence w.r.t. restriction. The definition should be modified in the case of output actions with variables, following the
Table 5: The ECCS definition of strong equivalence between programs.

\[ B_0 \sim B_1 \text{ iff } \forall k \geq 0 : B_0 \sim_k B_1, \text{ where } B_0 \sim_k B_1 \text{ is defined:} \]

- \( B_0 \sim_0 B_1 \) is always true
- \( B_0 \sim_{k+1} B_1 \) iff for all \( \Gamma(i = 0, 1) \):
  1. \( B_i \xrightarrow{\Gamma} B'_i \Rightarrow \exists B'_{i\oplus 1} : B_{i\oplus 1} \xrightarrow{\Gamma} B'_{i\oplus 1} \) and \( B'_i \sim_k B'_{i\oplus 1} \), if \( \Gamma = \tau, \lambda!v \)
  2. \( \forall v : [v/x]B'_i \sim_k [v/x]B'_{i\oplus 1} \), if \( \Gamma = \lambda?v \)
  3. \( \forall \alpha : [\alpha/x]B'_i \sim_k [\alpha/x]B'_{i\oplus 1} \), if \( \Gamma = \lambda!x \)

Consequences of the problem in Theorem 2 can also be detected in Theorem 4 as the following counter example shows.

\[ B_0 \equiv \gamma? | \delta!, \quad B_1 \equiv \gamma?.\delta! + \delta!.\gamma? \]

So not only was our original Theorem 4 hard to accept—it was false as stated!

However, the theorem holds under the proviso that \( \delta \) is not in the sort of \( B_0 \) and \( B_1 \), as correctly formulated for the \( \pi \)-calculus in [MPW89]. And with the modified definition of strong equivalence this is all what is needed for it to be a congruence w.r.t. restriction, as correctly noticed by and proved by Milner et al.

Notice that both definitions are late, in the sense that for input actions, the behaviour expressions \( B'_0 \) and \( B'_1 \) are required to be equivalent for all (program) instantiations of the bound variable. However, Milner et al. soon started looking at alternatives, see e.g. [MPW91] for definitions and characterisations of early and late bisimulations.

The next theorem collects some algebraic properties proved in the thesis accompanying [EN86] for ECCS. In the theorem \( g \) stands for a guard, i.e. \( \tau, a!e, a?x \) or \( a!x \).

**Theorem 6 (for programs)**
A binary relation $S$ is a strong bisimulation if both $S$ and its inverse are strong simulations.

Two agents $P$ and $Q$ are strongly ground bisimilar, written $P \sim Q$ iff there exists a strong bisimulation $S$ such that $P S Q$.

$P$ and $Q$ are strongly bisimilar, written $P \sim Q$, iff $P \sigma \sim Q \sigma$ for all substitutions $\sigma$.

Table 6: The π-calculus definition of strong simulation, strong ground bisimilarity and strong bisimilarity.

**Sum**

1. $B_0 + B_1 \sim B_1 + B_0$
2. $B_0 + (B_1 + B_2) \sim (B_0 + B_1) + B_2$
3. $B + \text{nil} \sim \text{nil}$
4. $B + B \sim B$

**Com**

1. $B_0 \mid B_1 \sim B_1 \mid B_0$
2. $B_0 \mid (B_1 \mid B_2) \sim (B_0 \mid B_1) \mid B_2$
3. $B \mid \text{nil} \sim B$
4. If $B_0, B_1$ are sums of guards then $B_0 \mid B_1 \sim$

\[
\sum\{g.(B_0' \mid B_1); g.B_0' \text{ a summand of } B_0\} +
\sum\{g.(B_0 \mid B_1'); g.B_1' \text{ a summand of } B_1\} +
\sum\{\tau.(B_0' [v/x] B_1'); \lambda v.B_0' \text{ a s. of } B_0, \lambda ?x.B_1' \text{ a s. of } B_1\} +
\sum\{\tau.(v/x)B_0' B_1'); \lambda x.B_0' \text{ a s. of } B_0, \lambda !v.B_1' \text{ a s. of } B_1\}
\]

**Res**

1. $(g.B) \setminus \alpha \sim \begin{cases} \text{nil} & \text{if } g = \alpha!v \\ g.B \setminus \alpha & \text{if } \alpha \not\in L(g) \end{cases}$
2. $B \setminus \alpha \sim B$, provided $\alpha \not\in L(B)$
3. $B \setminus \beta \sim B \setminus \beta \setminus \alpha$
4. $(B_0 + B_1) \setminus \alpha \sim B_0 \setminus \alpha + B_1 \setminus \alpha$
5. $(B_0 \mid B_1) \setminus \alpha \sim B_0 \setminus \alpha \mid B_1 \setminus \alpha$, provided $\alpha \not\in L(B_0) \cap L(B_1)$
Again a few comments are in place. First of all, our rationale at the time was to make sure that we had appropriately generalised versions of the laws from Milner’s axiomatisation of CCS, and we did not even consider the issue of completeness. Given that ECCS is parametrised in the language of data expressions, completeness would anyway be relative to the completeness of a proof system for data expressions in case such one would exist. However, Milner et al. had the superiority and insight to abandon the features of CCS irrelevant for mobility. With the simplifications they were able to give an equational theory for strong ground bisimulation complete for finite agents, based on versions of the laws from the Theorem above for ECCS, with one notable strengthening: Com(~(4) of Theorem 6 (“small” Expansion Theorem). Compared with the expansion theorem adopted by Milner et al. it lacks cases for guards like \((\lambda\alpha.B') \setminus \alpha\) corresponding to bound output actions in their terminology.

Furthermore, [EN86] also lifted strong equivalence to open expressions much in the same way as when Milner et al. introduced nonground bisimilarity. However, we could not lift our expansion theorem smoothly, a problem which was solved by Milner et al. by introducing the matching construct. More importantly, we only introduced our notion to allow ourselves to reason algebraically on our benchmark examples. Milner et al. initiated the study of nonground strong bisimilarity in its own right, e.g. proving that it forms a proper congruence for the \(\pi\)-calculus.

In addition an observational equivalence was suggested in [EN86], but the equivalence was only studied to the extent that certain desired properties of one of the benchmark examples (in particular the pushdown store) could be proven.

### 3.5 Expressiveness

As mentioned in the Preface, we had developed our own small set of benchmark examples to test expressiveness. One of them was based on some problems concerning CCS identified in chapter 9 of [Mil80], providing a phrase-by-phrase translation of an imperative programming language \(P\) into CCS. In this test of the expressiveness of CCS, Milner observed difficulties in handling certain programming language constructs, e.g. procedure call mechanisms with call-by-reference, and procedures admitting several concurrent
activations. In the following we recall how these problems were overcome by ECCS, mainly because this part of [EN86] at least to some extent could explain some of the complications of ECCS, certainly motivated originally by this particular benchmark.

Let us briefly repeat parts of the translation scheme of \( P \) into CCS from [Mil80]. The values of variables from \( P \) are kept in CCS registers:

\[
\text{LOC} \equiv \alpha? x. \fix X\langle y \rangle \langle \alpha? x.X(x) + \gamma! y.X(y) \rangle(x)
\]

\( \text{LOC}_Z \) is the register specially devoted to variable \( Z \) by defining:

\[
\text{LOC}_Z \equiv [\alpha_Z/\alpha][\gamma_Z/\gamma] \text{LOC}
\]

A value for \( Z \) is stored via \( \alpha_Z \) and read via \( \gamma_Z \), and the scope of a variable is limited by restricting with \( L_Z = \{\alpha_Z, \gamma_Z\} \).

\( \pi \)-calculus The translation of \( P \)-expressions \( E ([E]) \) is such that it delivers the result via \( \varrho \), introducing the notation:

\[
B_0 \text{ result } B_1 \text{ denoting } (B_0 \mid B_1) \setminus \varrho
\]

And finally, a translated program \([C]\) signals its completion via \( \delta \) where- upon it “dies”. It is therefore convenient to define

\[
\text{done} \equiv \delta!.\text{nil}
\]

and

\[
B_0 \text{ before } B_1 \equiv ([\beta/\delta]B_0 \mid \beta?.B_1) \setminus \beta
\]

for a \( \beta \not\in L(B_0) \cup L(B_1) \).

Now we suggested in [EN86] to translate a procedure declaration with call-by-value and call-by-result parameters into ECCS as follows

\[
[\text{PROC } G(\text{VALUE } X, \text{RESULT } Y) \text{ is } C_G] = \fix X_G\langle (\alpha_G!\lambda.(X_G \mid \text{LOC}_X \mid \gamma? X.\alpha X!X.\text{[C_G]}) \\text{ before } \gamma? y.\lambda! y.\text{nil}) \setminus L_X \setminus L_Y \rangle \setminus \lambda
\]

and to translate a corresponding call as

\[
[\text{CALL } G(E, Z)] = [E] \text{ result } (\varrho? x.\alpha_G? y.y!x.y? z.\alpha_Z!z.\text{done})
\]

The problem identified by Milner in handling concurrent activations was in this translation solved by letting the activated procedure return a communication link private to the caller and that particular activation. Our solution to the problem of procedures with call-by-reference parameters consisted of translating procedure declarations as
\[
\begin{align*}
[ \text{PROC } G(\text{REF } Z) ] &= \text{C}_G \\
&= \text{fix } X_G \langle (\alpha_G ! \lambda. \langle X_G \mid (\lambda ? z_\alpha . \lambda ? z_\gamma . [z_\alpha / \alpha_Z] [z_\gamma / \gamma_Z] [C_G] \rangle \text{ before } \lambda ! . \text{nil}) \rangle \rangle \lambda
\end{align*}
\]

and corresponding calls as
\[
[ \text{CALL } G(Y) ] = \alpha_G ? y . y ! \alpha_Y . y ! ? \gamma_Y . y ? \text{. done}.
\]

The issue of expressiveness of the \( \pi \)-calculus was addressed much more seriously Milner et al. They showed how the effect of process-passing could be achieved by link-passing and gave encodings of values and various data structures. More notably they devised encodings of combinator calculi and the lazy \( \lambda \)-calculus.

Given the data encodings of Milner et al., they would be able to deal with all of our benchmark examples and in particular the problems of the translation could be solved in the \( \pi \)-calculus exactly as in ECCS.

4 Conclusion from 1986

ECCS as presented here is one attempt at a smooth extension of CCS satisfying the goals outlined in the introduction. In the process of defining ECCS we have considered a great number of alternatives—slowly converging to ECCS in its present form. We feel, and we hope the reader feels the same, that ECCS is reasonably in line with the elegance of Milner’s CCS—at least this has been one of our main guidelines in the process. We also feel there are some sound ideas underlying the calculus of ECCS, but we certainly do not claim that ECCS is the end product. There are still aspects with which we feel uneasy, and let us just mention a few.

One has to do with the fact that in ECCS, as presented here, only one value may be communicated at a time. As long as we are only concerned with normal data-values, it is obvious how to extend the calculus to allow tuples of values (as in CCS). But for label values the situation is not quite so clear. At least, it requires some thought how to formulate “multi-change of scope in connection with single communication”. We have chosen to present ECCS without going into these problems. Also, we have deliberately chosen not to consider the problems involved in generalising CCS to allow passing of processes.

Another slightly displeasing thing about ECCS is the fact that labels are somehow considered both as variables (bound by restriction) and values (to be substituted for label variables bound by input commands).
Furthermore, ECCS obviously needs to be tried out on more challenging examples than the small toy problems we have considered in this paper.

Despite considerations like the above, which indicate that ECCS is maybe not yet “quite right”, we are confident that ECCS represents a step on the right track in the process of solving the problems we set out to solve.

5 Final Remarks

Looking back at our Conclusion from 1986, the problem of sending multiple links or channels was successfully solved by Milner in a paper on the polyadic $\pi$-calculus [Mil91] through the use of structural congruence. Essentially the notion of $\alpha$-convertibility is built into the definition of the action relations by adding a rule ensuring that a behaviour expression can do whatever action an $\alpha$-equivalent expression can do, where we had this as a derived property of strong equivalence, i.e. that $\alpha$-convertable programs are strong equivalent. After the appearance of the chemical abstract machine [BB90] Milner presented the semantics of the $\pi$-calculus via structural congruence in the paper “Functions as Process” [Mil90] (where the main issue was the encoding possibilities w.r.t. the $\lambda$-calculus) and showed that the new and by far much simpler formulation of the semantics was consistent with the original operational semantics. The idea was soon to find footing in the community, see e.g. Crasemann [Cra92] for an extensive use.

Regarding our comments on the dual role of labels, Milner et al. took (as commented in section 3.1) it to its logical conclusion and identified labels and label variables.

And finally, on the issue of challenging examples, Milner et al. considered a number additional examples most notably encodings of $\lambda$-calculi. Later additional “natural” mobility examples came up—cf. [Mil91, Wal91, OP92, Jon93]

But all this is just a small part of the impressive amount of research which has been done over the years on the $\pi$-calculus by Milner and many other researchers, and which is beyond the scope of this note.

Acknowledgement

We would like to thank Paola Quaglia, the editors, and two anonymous referees for valuable comments on various drafts of this note.
References


