INTRODUCTION.

The aim of this paper is to show a technique for getting an operational semantics for the language of Distributed Processes (DP) introduced in /BH/, by means of a translation into the language of behaviour expressions in CCS, the Calculus of Communicating Systems developed in /M/.

The essential idea is as follows. Guided by an intuitive understanding of the semantics of DP, as exposed in /BH/, we can translate DP terms into CCS terms (Section 1); then, if P is a DP term and \( [P] \) denotes its translation, to P is associated a semantics, which is the semantics of \( [P] \) in CCS, obtained from a system of derivations \( \mathcal{A} \). This semantics is not very manageable, since it is expressed in a kind of low level language. But from \( \mathcal{A} \) we can deduce a simpler system of derivations \( \mathcal{B} \) acting on CCS terms which are the translation of DP terms; thus, forgetting the CCS structure, and using a simplified notation, we get a system of derivations on DP terms (Section 2). This system provides a way of defining an operational semantics which agrees completely with a semantics in CCS, to be made precise in the paper (Section 3).

One important point is that this technique relies on a quite general principle; and indeed it has been already applied to the case of CSP (see /AZ/; for a related paper see /HLP/).

From another viewpoint, if we accept as good the obtained operational semantics of Section 2, the results of Section 3 show that the translation in Section 1 is correct. Moreover the translation seems to give good hints and directions for a correct implementation.

Throughout the paper, we assume the reader familiar with the basic concepts and notations about CCS as in /M/ and DP as in /BH/.

1. TRANSLATION OF DP PROCESSES.

1.1 Assumptions. We recall here in outline the basic ideas of DP:
- a program consists of a fixed number of concurrent processes that start simultaneously and exist forever.
- each process can access its own variables only (there are no common variables).
- processes communicate only via common procedures.
- a process defines some procedures and an initial statement.
- the execution of a process begins with the execution of the initial statement; the execution continues until the statement either terminates or arrives at a "guarded

This work has been partially supported by a grant of "Progetto Finalizzato Informatica-Pl-CNET" of CNR (Consiglio Nazionale delle Ricerche)-Italy.
(when or cycle statement) where all the boolean guards are false; then another
operation starts (as the result of an external request) and an interleaving begins of
the initial statement and external requests.

There is a common agreement on the following two assumptions: recursive procedures are
not allowed and a process cannot call its own procedures (see /BH/ and, e.g., 
/RDKR/).
In this paper we shall also assume that a procedure can be accessed by only one process
at a time, so that, in the case of several "contemporary" calls, one process is served,
while the others have to wait. This is a restriction with respect to the DP originally
proposed. The motivation is purely technical (see /M/ pg.134): in the version of CCS which
we refer to no easy way can be found for allowing simultaneous activation of an unbound-
ed number of copies of a procedure (it would be easy instead to allow for a fixed
maximum number). The easiest way for circumventing this difficulty, i.e., using a "return
link", requires passing labels of channels as values, which is not allowed in CCS (see
/M/ Ch.9). But it should be said that some work is in progress for an improved version
of CCS without this limitation.

Finally our translation assumes that a process is an agent which exists forever, as
in /BH/ i.e. we do not make any assumption about termination of programs (contrary to
the way taken in /RDKR/). We could have taken the other approach, but in that case the
translation should be different; that is the way taken in CSP and the reader is referred
to /AZ/ for a translation dealing with this situation (Obviously, if processes exist
forever, the results of a program are to be obtained through procedure calls from an
external agent, say a printer).

Now let us recall some basic clauses of the DP syntax (lower level clauses will be
be given when defining the translation).

<program> ::= <process> [ | <process> ]
<process> ::= <process name> < declaration> { <procedure> } <initial statement>
<procedure> ::= <proc name> ( { <input param> } | <output param> ) <declaration>
<statement> ::= <assignment> | <procedure call> | <if statement> | <do statement>
<empty statement>

We denote by V the set of values of the expressions. In what follows S will range over
statements; P, Q over process names; R, T over procedure names; E over expressions; B over
boolean expressions; D over declarations; X, Y over variables; x, y, u, v, w over values.

1.2 Preliminary comments. Important remark: in what follows strong equivalence between
behaviour expressions will be denoted by = instead of ~ as in /M/ pg. 77.

In what follows we give the translation of the DP language into CCS; we recall that
CCS is a model for concurrent computation where the only actions are communications
between elementary agents; hence the effect of this translation is to represent every
step in the execution of a DP program by communications of low level components. To get
just the flavour of the translation, let us consider the case of the assignment.
A variable X can be represented by a register of sort \( \{a, y, x\} \), defined as follows:

\[
\text{REG}(x) \triangleq \begin{cases} 
\text{if } x=0 \text{ then } \alpha.y.\text{REG}(y) \text{ else } \gamma.y.\text{REG}(y) + \gamma.x.\text{REG}(x) 
\end{cases} 
\]

(see /M/ pg.127)

We will abbreviate \( \text{REG}(x)[\alpha.x \gamma.y \alpha.y] \) by \( \text{REG}_x(x) \).

Here a register of content x is an agent which can both receive a new value y on the
line \( \alpha \) and answer an external request sending its content x. These communication
capabilities are expressed formally by means of derivation rules:

\[
\text{REG}(x) \xrightarrow{\alpha} \text{REG}(v) \text{ for every } v \text{ in the value set of } y.
\]