Theoretical Computer Science 33 (1984) 45–63
North-Holland

PARAMETRIC CHANNELS VIA LABEL EXPRESSIONS IN CCS*

Egidio ASTESIANO and Elena ZUCCA
Istituto di Matematica, Università di Genova, 16132 Genova, Italy

Abstract. CCS, the Calculus of Communicating Systems devised by Milner, has proved extremely successful for providing, via translation, a sound mathematical basis for a wide class of concurrent languages. Nevertheless, as it stands, it suffers from a limitation: it is impossible to determine at run time the channel on which to send or receive a communication. There are various possibilities for giving CCS such ability, but the price can be a drastic change in the language and the theory of CCS.

Here we present a simple solution to this problem, which keeps language and theory almost unchanged, at least in fundamental aspects: the essential idea is to extend CCS by allowing (channel-) label expressions. We also show various applications of this extended version of CCS.

1. Introduction

1.1. Translating into CCS

In [7], Milner has presented a Calculus of Communicating Systems (abbreviated CCS) which consists of:

(a) a language, the language of behaviour expressions;

(b) a set of derivation rules which define a labelled relation on behaviour expressions;

(c) a semantics for the behaviour programs (i.e., behaviour expressions without free variables) defined by some equivalences, based on the idea of observation: i.e., two programs are equivalent if and only if their observable behaviour is equivalent in some sense.

In [7, 2, 1, 6] techniques are exhibited for obtaining an operational semantics for concurrent languages by means of a translation into CCS. We illustrate here the basic ideas; the reader acquainted with these topics can pass directly to Section 1.2.

The most classical example is a memory register,

$$\mathrm{REG}_X(n) \Leftarrow \alpha_X m.\mathrm{REG}_X(m) + \bar{\gamma}_X n.\mathrm{REG}_X(n),$$

$$\mathrm{LOC}_X \Leftarrow \alpha_X m.\mathrm{REG}_X(m).$$

A register X which contains the integer n is represented by a behaviour program $\mathrm{REG}_X(n)$, recursively defined, which has two communication capabilities: it can

* Work partially supported by CNR-PFI-PI CNET and by a Grant of Ministero Pubblica Istruzione.

receive on the input port α a new integer value m and then become $\mathrm{REG}_X(m)$ or it can send through the outport port $\bar{\gamma}$ the contained value n and restore itself. LOC_X denotes a register initially empty.

Now consider the translation of some typical programming constructs (translation is denoted by the brackets $[\![\ldots]\!]$),

$[\![X := 0]\!] = ([\![0]\!] \mid \rho m.\bar{\alpha}_X m.\bar{\varepsilon}.\mathrm{NIL})\backslash\rho$, where $[\![0]\!] = \bar{\rho}0.\mathrm{NIL}$;

$[\![C; C']\!] = [\![C]\!]$ before $[\![C']\!]$, where B before B' is a short notation for $(B[\beta/\varepsilon] \mid \beta.B')\backslash\beta$ (β new), for any behaviour expression B, B'.

The behaviour program which translates an assignment consists of the parallel composition (denoted by $\mid$) of two programs: the first one is the translation of the expression 0 and just sends the integer value 0 and then dies (NIL is the program which has no capabilities); the second one is a program which receives an integer value m at the port ρ, sends it through the output port $\bar{\alpha}_X$ to the register X shown above and then dies. The port $\bar{\varepsilon}$ just sends a control signal which is needed to manage sequential composition. This is shown in the second example: the translation of any command C must end with the sending of a signal which allows the following command C' to start. The 'restrictions' $\backslash\rho$ and $\backslash\beta$ denote that the ports ρ, β can be used only 'inside' the restriction operator $\backslash$. The 'relabelling' $[\beta/\varepsilon]$ denotes that every occurrence of ε in $[\![C]\!]$ has to be replaced by a never used label β; that ensures that every start signal will be received only by the correct command.

For a full description of CCS and of the technique of translating see [7, 2, 1, 6]. However, it should be clear from the examples above that the basic idea of translating into CCS is to describe any action in terms of a number of elementary synchronized communications through 'complementary' ports (like α and $\bar{\alpha}$). This approach is very simple and elegant, but it is not sufficient to describe all communication constructs found in concurrent languages.

1.2. Parametric channels

An inadequacy of the original version of CCS, noticed by Milner himself [7, pp 134–135], is the impossibility of describing situations where a process must decide which are its communication ports at run time, since in every CCS program the ports are statically determined and finite in number. For example, we cannot translate into CCS a language where a procedure can be called simultaneously from any number of processes, causing the parallel execution of more 'instances' of the procedure. Let us recall and state the problem. Consider the translation of a typical procedure (assume it is not recursive, for the moment), $[\![\textbf{proc}\ G(X:\textbf{in}\ Y:\textbf{out})\ \mathrm{body}]\!] = g$, where g is a behaviour identifier recursively defined as follows:

$$g \Leftarrow (\mathrm{LOC}_X \mid \mathrm{LOC}_Y \mid \alpha_G x.\bar{\alpha}_X x.[\![\mathrm{body}]\!] \text{ before } \gamma_Y y.\bar{\gamma}_G y.g)\backslash L_X \backslash L_Y.$$

(For any register X, $\backslash L_X$ is a shortened notation for $\backslash\alpha_X \backslash\gamma_X$.)

The translation of the procedure G consists of the parallel composition of the registers X, Y, initially empty, and of a program which, after receiving an input value

x from some calling process at the port α_G, stores it in X through the port $\bar{\alpha}_X$, then executes the body of G, sends to the calling process the content of Y on $\bar{\gamma}_G$ and finally restores g so that the procedure can be called again.

The translation of a call will be

$$[\![G(\text{expr}, Z)]\!] = ([\![\text{expr}]\!] \mid \rho x.\bar{\alpha}_G x.\gamma_G z.\bar{a}_Z z.\bar{\varepsilon}.\text{NIL})\backslash\rho,$$

where we assume that the program $[\![\text{expr}]\!]$ evaluates expr and sends the obtained value through $\bar{\rho}$. Thus the translation of a call is a behaviour program that, after computing the value x of expr, sends it to the procedure G through $\bar{\alpha}_G$, waits for receiving the output parameter z on γ_G and finally stores it in the register Z through $\bar{\alpha}_Z$.

This translation is fully satisfactory if the procedure allows only one execution at a time, i.e., other possible calls must wait on $\bar{\alpha}_G$ until g has restored itself after an execution. At first sight, we might hope to allow for concurrent activations of G by making g restore itself directly after receiving its argument, $[\![\textbf{proc } G(X:\textbf{in}; Y:\textbf{out}) \text{ body}]\!] = g$, where

$$g \Leftarrow \alpha_G x.(g \mid (\text{LOC}_X \mid \text{LOC}_Y \mid \bar{\alpha}_X x.[\![\text{body}]\!] \text{ before } \gamma_Y y.\bar{\gamma}_G y.\text{NIL})\backslash L_X \backslash L_Y).$$

Now the restored g can be activated immediately after the first, and run concurrently with it. But we cannot be sure that every g will return its result to the correct calling process, because all the calling processes will be waiting on γ_G and then any one of them could communicate with g which is waiting on $\bar{\gamma}_G$, following the rule which handles communication in CCS, based only on the 'complementarity' of the ports (like α and $\bar{\alpha}$).

What we want to do in this paper is to give CCS the ability to have 'parametric channels' (from another point of view, to 'pass labels'), in order to model situations where the partner of a communication is dynamically determined. To this end in Section 2 we describe an extension of CCS, called CCS/PC, which admits parametric channels; more precisely, we replace labels of CCS (like α, $\bar{\alpha}$) by 'label expressions' indicated by $\Phi, \Phi_1, \ldots, \Phi_n$ and which can be evaluated to label constants, provided they are closed, i.e., without free variables. Recalling that the behaviour of a CCS program is determined modulo the association of an evaluation system for the value expressions, we can say that now the behaviour of a CCS/PC program is determined modulo the association of an evaluation system both for value and label expressions. In Sections 2.1 and 2.2 we describe syntax and semantics of CCS/PC; in Section 3 we present some applications, notably the handling of concurrent procedure calls and the translation into CCS/PC of a fragment of the metalanguage introduced by Bjørner and Folkjaer [3] for specifying formally the semantics of ADA tasking [4]. Finally, in Section 4 we outline an extension for dealing with other constructs and give some hints towards further work.

Note that the paper is not self-contained; we assume Milner's book [7] as a standard reference, in order to avoid unnecessary repetitions of technical statements. To that book we obviously refer for motivations.

More recently, in [8], Milner has developed a new theory of communicating processes, called SCCS, without value passing and with only four combinators and recursion without parameters, in which the binary operator + is replaced by infinite indexed sums. We will give in the last section some comments on the relationship between our CCS/PC and SCCS.

2. CCS/PC

2.1. The language

Value expressions. As in pure CCS, we use E, possibly indexed, to range over the set of value expressions. We do not specify their syntax but only assume that value expressions can contain variables $x, y, \ldots$. As we shall see later, when dealing with semantics, every value expression without free variables can be evaluated, following an evaluation system, to a unique constant symbol v denoting a value in a set of values. We will also allow tuples $(E_1, \ldots, E_n)$ of value expressions.

Label constants and relabellings. We assume a fixed set Δ of positive label constants, ranged over by $\alpha, \beta, \gamma, \ldots$, and a set $\bar{\Delta}$ of negative label constants, ranged over by $\bar{\alpha}, \bar{\beta}, \bar{\gamma}, \ldots$, disjoint from Δ and in bijection with it, the bijection being $\alpha(\in \Delta) \to \bar{\alpha}(\in \bar{\Delta})$. Our set of label constants is $\Lambda = \Delta \cup \bar{\Delta}$, together with τ. We use λ to range over Λ; $\mu, \nu, \ldots$ to range over $\Lambda \cup \{\tau\}$ and A to range over non-empty subsets of Δ. If $A \subseteq \Delta$, then we define $\bar{A} = \{\bar{\alpha} \mid \alpha \in A\}$. A relabelling $S: \Lambda \to \Lambda$ is a bijection which respects complements (i.e., $S(\bar{\alpha}) = \overline{S(\alpha)}$ for every $\alpha, \bar{\alpha} \in \Lambda$).

Label expressions. We use Φ, possibly indexed, to range over the set of positive label expressions, whose syntax, as for the value expresssions, is not specified. We only assume that positive label constants are positive label expressions and that positive label expressions can contain variables $x, y, \ldots$. The intended meaning, as we shall better see later, is that every positive label expression without free variables can be evaluated, following an evaluation system, to a unique positive label constant. For every positive label expression Φ, $\bar{\Phi}$ will denote the corresponding negative label expression: the intended meaning is that if Φ evaluates to α, then $\bar{\Phi}$ evaluates to $\bar{\alpha}$.

Behaviour identifiers p. We assume that a collection of identifiers is given, each one having a preassigned arity $n(p)$, the number of value parameters. We assume that the meaning of such identifiers is given by a clause

$$p(x_1, \ldots, x_{n(p)}) \Leftarrow P_p,$$

where $x_1, \ldots, x_{n(p)}$ are distinct variables and P_p is a behaviour expression, defined below. We assume moreover that the behaviour identifiers are guardedly well-defined [7, p. 72].

Behaviour expressions. A behaviour expression, is defined inductively as follows:

(i) NIL is a behaviour expression;

(ii) if P, P' are behaviour expressions, then

$$P+P', \quad \Phi x_1, \ldots, x_n.P, \quad \bar{\Phi}E_1, \ldots, E_n.P, \quad \tau.P, \quad P|P', \quad P\backslash A,$$

$$P[S], \quad p(E_1, \ldots, E_{n(p)}), \quad \textbf{if } E \textbf{ then } P \textbf{ else } P'$$

are behaviour expressions.

In what follows we shall use $B\{E_1/x_1, \ldots, E_n/x_n\}$ to denote the result of substituting the expression E_i for the variable x_i ($1 \leq i \leq n$) at all its free occurrences within B. Sometimes we shall abbreviate tuples of variables and expressions as $\tilde{x}$ and $\tilde{E}$, and write a substitution as $B\{\tilde{E}/\tilde{x}\}$.

The assumptions on free variables are as follows:

FV(NIL) is the empty set;

$\mathrm{FV}(P+P') = \mathrm{FV}(P) \cup \mathrm{FV}(P')$;

$\mathrm{FV}(\Phi x_1, \ldots, x_n.P) = \mathrm{FV}(P) - \{x_1, \ldots, x_n\} \cup \mathrm{FV}(\Phi)$;

$\mathrm{FV}(\bar{\Phi}E_1, \ldots, E_n.P) = \mathrm{FV}(P) \cup \bigcup_{i=1}^{n} \mathrm{FV}(E_i) \cup \mathrm{FV}(\bar{\Phi})$;

$\mathrm{FV}(\tau.P) = \mathrm{FV}(P)$; $\quad \mathrm{FV}(P|P') = \mathrm{FV}(P) \cup \mathrm{FV}(P')$;

$\mathrm{FV}(P\backslash A) = \mathrm{FV}(P)$; $\quad \mathrm{FV}(P[S]) = \mathrm{FV}(P)$;

$\mathrm{FV}(p(E_1, \ldots, E_{n(p)})) = \bigcup_{i=1}^{n(p)} \mathrm{FV}(E_i)$;

$\mathrm{FV}(\textbf{if } E \textbf{ then } P \textbf{ else } P') = \mathrm{FV}(E) \cup \mathrm{FV}(P) \cup \mathrm{FV}(P')$.

The meaning and the use of free variables is as usual. A program is a behaviour expression without free variables.

Comparing these definitions to the ones given in [7, pp. 66–67], the reader can realize that our behaviour expressions are just CCS behaviour expressions where we allow label expressions. Moreover, we admit restriction over a possibly infinite set of labels (i.e., we use $\backslash A$, A being a set of positive labels, instead of $\backslash\alpha$ as in [7]); that is needed here because the set of labels of a behaviour expression is now potentially infinite. An analogous operator can be found in SCCS [8]. We will not give a fixed syntax for the sets of positive labels: that corresponds to the standard treatment of relabelling functions here and in the original CCS [7]. Clearly, in every instantiation of their use a finite effective presentation will be adopted. As a standard notation, if A has only one element α, then we will write just $B\backslash\alpha$.

For an intuitive understanding of the difference between CCS and CCS/PC, it can be useful to consider the trees corresponding respectively to CCS and CCS/PC programs. We recall that in [7, Ch. 6] a natural way is presented of representing behaviour programs as a special kind of trees, called Communication Trees (abbreviated CTs). We refer to [7] for a full description. Milner has already pointed out [7, p. 97] that only a small subclass of CTs are expressible as CCS programs. A larger

class is expressible as CCS/PC programs; we do not give here a formal definiti of this class, but describe it by an example.

Consider the CT in Fig. 1; it is not expressible as a CCS program, but is expressit as the CCS/PC program $\alpha x.\overline{\gamma(x)}.\mathrm{NIL}$, where x is a variable with values in $\mathbb{N}$ a the label expression $\gamma(n)$ evaluates to γ_n $(n \geq 0)$.

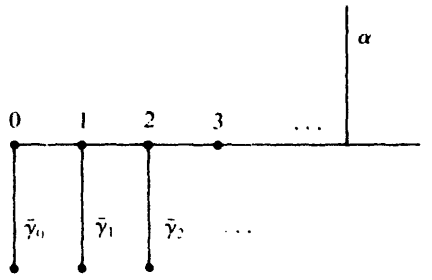

Fig. 1.

Notice also that there is a larger class of CTs which are not even expressible CCS/PC programs; for example, the one in Fig. 2.

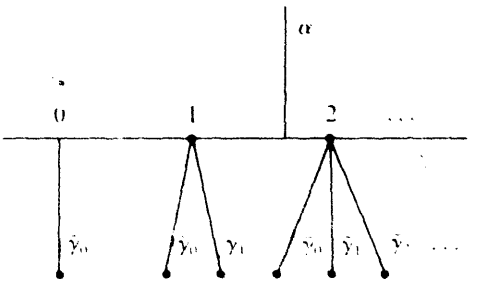

Fig. 2.

This CT would correspond to allowing a behaviour expression which is a para meterized sum, and written $\alpha n.\sum_{i=0}^{n} \bar{\gamma}_i.\mathrm{NIL}$. This is neither possible in CCS nor i CCS/PC (parameterized sum is not allowed). At the end of the paper we shall se some further extensions, corresponding to slightly different assumptions on the labe expressions in CCS/PC, which include the above CTs and many more, while no producing any important change in the theory.

2.2. *Semantics by derivations and equivalences*

In [7, pp. 69–71], Milner defines a binary relation $\rightarrow^{\mu v}$ over behaviour expressions for each $\mu \in \Lambda \cup \{\tau\}$ and value v (of type appropriate to μ). $B \rightarrow^{\mu v} B'$ may be reac 'B produces (or can produce) B' under μv'. The relations $\rightarrow^{\mu v}$ are defined by induction on the structure of the behaviour expressions. Such relations, whicl altogether define a labelled transition system, here also called a derivation system are the basis for the definition of semantic equivalences.

Here we extend this system to CCS/PC. We assume that a transition system is given for evaluating the value expressions and the label expressions; the arrow $\to_E$ and its transitive closure $\to_E^*$ denote the transitions and thus for every closed (i.e., without free variables) value expression E and label expression Φ, there is a unique value v and a unique label α s.t. $E \to_E^* v$ and $\Phi \to_E^* \alpha$. (The assumption on uniqueness of values is only for simplicity; at the end of the paper we shall see some easy extensions.) According to our previous convention, if $\Phi \to_E^* \alpha$, then $\bar{\Phi} \to_E^* \bar{\alpha}$. Now, before defining the derivation system for behaviour expressions, let us recall that in [7] a closed value expression is considered identical with its value. Formally, the rules

$$\frac{\tilde{E} \to_E^* \tilde{v}}{\bar{\alpha}\tilde{E}.B \to \bar{\alpha}\tilde{v}.B} \qquad \frac{E \to_E^* v}{\textbf{if } E \textbf{ then } B_1 \textbf{ else } B_2 \to \textbf{if } v \textbf{ then } B_1 \textbf{ else } B_2}$$

(where $\tilde{E}$ and $\tilde{v}$ denote tuples of expressions and values), are implicitly assumed. An equivalent formulation of the same concept is that an inference rule such as

$$\left[\frac{\tilde{E} \to_E^* \tilde{v}}{\bar{\alpha}\tilde{E}.B \to^{\bar{\alpha}\tilde{v}} B}\right] \text{ is written as an axiom } \bar{\alpha}\tilde{v}.B \to^{\bar{\alpha}\tilde{v}} B.$$

We shall stick to the same convention; hence, instead of writing, for example,

$$\frac{\Phi \to_E^* \alpha,\ \tilde{E} \to_E^* \tilde{v}}{\bar{\Phi}\tilde{E}.P \to^{\bar{\alpha}\tilde{v}} P} \quad \text{we write directly } \bar{\alpha}\tilde{v}.P \to^{\bar{\alpha}\tilde{v}} P.$$

The remarks above are in the spirit of Plotkin's structural operational semantics (see [9]), which is the methodology behind the semantics by derivations of CCS.

Thus, perhaps not surprisingly, with the above convention the schemes of axioms and rules defining the derivations (labelled transition system) in CCS/PC are formally the same as in CCS [7, pp. 69–71], except that for the extended restriction mentioned above. We list the rules here; the explanatory comments are as in [7]. P ranges over behaviour expressions in CCS/PC; as in [7] we write also μv, λv for $\mu\tilde{v}$, $\lambda\tilde{v}$.

(Sum) (1) $\dfrac{P_1 \to^{\mu v} P_1'}{P_1 + P_2 \to^{\mu v} P_1'}$, (2) $\dfrac{P_2 \to^{\mu v} P_2'}{P_1 + P_2 \to^{\mu v} P_2'}$;

(Act) (1) $\alpha x_1, \ldots, x_n.P \xrightarrow{\alpha(v_1, \ldots, v_n)} P\{v_1/x_1, \ldots, v_n/x_n\}$,

(2) $\bar{\alpha}v_1, \ldots, v_n.P \xrightarrow{\bar{\alpha}(v_1, \ldots, v_n)} P$,

(3) $\tau.P \to^{\tau} P$;

(Com) (1) $\dfrac{P_1 \to^{\mu v} P_1'}{P_1 | P_2 \to^{\mu v} P_1' | P_2}$, (2) $\dfrac{P_2 \to^{\mu v} P_2'}{P_1 | P_2 \to^{\mu v} P_1 | P_2'}$,

(3) $\dfrac{P_1 \to^{\lambda v} P_1' \quad P_2 \to^{\bar{\lambda} v} P_2'}{P_1 | P_2 \to^{\tau} P_1' | P_2'}$;

(Res) $$\frac{P \to^{\mu v} P'}{P\backslash A \to^{\mu v} P'\backslash A} \quad \mu \notin A \cup \bar{A};$$

(Rel) $$\frac{P \to^{\mu v} P'}{P[S] \to^{S(\mu v)} P'[S]};$$

(Ide) $$\frac{P_p\{v_1/x_1, \ldots, v_{n(p)}/x_{n(p)}\} \to^{\mu v} P'}{p(v_1, \ldots, v_{n(p)}) \to^{\mu v} P'};$$

(Con) (1) $$\frac{P_1 \to^{\mu v} P_1'}{\textbf{if true then } P_1 \textbf{ else } P_2 \to^{\mu v} P_1'},$$

(2) $$\frac{P_2 \to^{\mu v} P_2'}{\textbf{if false then } P_1 \textbf{ else } P_2 \to^{\mu v} P_2'}.$$

Note that the derivations are defined on programs; the consistency of the definitions above is assured by the following lemma, which relies on the definition of free variables and the convention above (closed expression ≡ denoted value/label).

Lemma 2.1. (i) *A* CCS/PC *behaviour expression is a program iff it has one of the following forms:*

$$\text{NIL}, \quad P_1 + P_2, \quad \alpha x_1, \ldots, x_n.P, \quad \bar{\alpha} v_1, \ldots, v_n.P', \quad \tau.P', \quad P_1 \mid P_2,$$

$$P'\backslash A, \quad P'[S], p(v_1, \ldots, v_{n(b)}), \quad \textbf{if } v \textbf{ then } P_1 \textbf{ else } P_2,$$

where P_1, P_2, P' *are programs and the only free variables in* P *are* $x_1, \ldots, x_n$.

(ii) *If* P *is a* CCS/PC *program and* $P \to^{\mu v} P'$, *then* P' *is also a program.*

Proof. (i) Obvious from definitions.

(ii) By induction on the length of the inference which ensures $P \to^{\mu v} P'$. □

As for CCS, we can now proceed to define various equivalences on behaviour programs and thus, extending by closure, on behaviour expressions. In [7] three equivalences are considered:

(1) direct, denoted by ≡;

(2) strong, denoted by ∼;

(3) observational, denoted by ≈.

Because of Lemma 2.1 and the fact that the derivation rules are the same, we can assume the same definitions for CCS/PC [7, pp. 74, 70, 99]. But, more importantly all the propositions in [7] on equivalences of behaviour programs in CCS extend, essentially with the same proofs, to CCS/PC; there is only a minor point to be explicitly noted and adjusted. Since some of the propositions involve the notion of sort and names of a behaviour expression, we have to extend that notion to CCS/PC behaviour expressions. That extension can be done in a straightforward way.

Recall that the sort of a behaviour expression in CCS is the set of its positive and negative channel labels (formally defined in [7, p. 68]). Clearly, in CCS the sort of a behaviour expression is statically determined, since the channel labels are constant expressions: for a CCS/PC behaviour expression we have to consider as sort the set of all positive and negative channel labels to which the label expressions can evaluate, in the underlying system for the expressions that we have implicitly assumed in every CCS calculus.

Assuming that $\tilde{x}$ denotes the tuple of the free variables of Φ, define $L(\Phi) = \{\lambda \mid \Phi\{\tilde{v}/\tilde{x}\}$ evaluates to λ, for some tuple of values $\tilde{v}\}$ and $L(\bar{\Phi}) = \{\bar{\lambda} \mid \lambda \in L(\Phi)\}$.

Then the sort $L(P)$ of a CCS/PC behaviour expression P can be defined inductively as in [7, p. 68], with the following changes:

$$L(\Phi x.P) = L(P) \cup L(\Phi), \qquad L(\bar{\Phi}\tilde{E}.P) = L(P) \cup L(\bar{\Phi}),$$

$$L(P\backslash A) = L(P) - (A \cup \bar{A}).$$

In some statements the following notations will be used: $\text{name}(\alpha) = \text{name}(\bar{\alpha}) = \alpha$; if L is a sort, then $\text{names}(L) = \{\text{name}(\lambda) \mid \lambda \in L\}$; if g denotes a guard (i.e., $\Phi\tilde{x}$, $\bar{\Phi}\tilde{E}$ or τ), then $\text{names}(g) = \text{names}(L(\Phi))$.

Then, assuming that the variables ranging over CCS behaviour expressions now range over CCS/PC behaviour expressions, we can for example state the following theorem about CCS/PC programs.

Theorem 2.2. *Theorems* 5.3 (*about direct equivalences*), 5.4 *and* 5.5 (*about strong congruences*) *of* [7] *hold in* CCS/PC.

Proof. The proof follows by using Lemma 2.1, as in [7]. □

As a clarifying example, recall the assertion (Res) ~ (1), of [7, Theorem 5.5],

$$B\backslash\alpha \sim B, \quad \alpha \notin \text{names}(L(B)),$$

and consider the behaviour program $P = \beta f.\bar{f}v.\text{NIL}$, where f is a variable with values in Δ (the set of positive labels). Then the above assertion cannot be applied to P, since $\bar{f}$ can evaluate to $\bar{\alpha}$ and hence $\alpha \in \text{names}(L(P))$ (and indeed clearly $P\backslash\alpha \not\sim P$).

Again as in [7] we can extend the result to behaviour expressions, first defining direct and strong equivalences,

$$P_1 \equiv P_2 \text{ iff for all } \tilde{v},\ P_1\{\tilde{v}/\tilde{x}\} \equiv P_2\{\tilde{v}/\tilde{x}\},$$

$$P_1 \sim P_2 \text{ iff for all } \tilde{v},\ P_1\{\tilde{v}/\tilde{x}\} \sim P_2\{\tilde{v}/\tilde{x}\},$$

where $\tilde{x}$ is the tuple of the free variables occurring in P_1 or P_2 or both. Hence we can say, for example, as the intuition clearly suggests, that $\bar{\Phi}(x+1).\text{NIL}|\text{NIL}$ is strongly equivalent to $\bar{\Phi}(x+1).\text{NIL}$ for any label expression Φ.

For another example, assume that f is a variable with values in Δ; then $(\bar{f}v.\text{NIL})\backslash\alpha$ is not strongly equivalent to $\bar{f}v.\text{NIL}$, since substituting α for f, we have $(\bar{f}v.\text{NIL})\backslash\alpha\{\alpha/f\} = (\bar{\alpha}v.\text{NIL})\backslash\alpha$ which is strongly equivalent to NIL, while

$\bar{f}v.\mathrm{NIL}\{\alpha/f\} = \bar{\alpha}v.\mathrm{NIL} \not\sim \mathrm{NIL}$. With the above definitions, we can generalize the previous results to behaviour expressions (without proof, which is routine).

Theorem 2.3. *Theorem* 5.7 *of* [7] *holds for* CCS/PC *behaviour expressions.*

Analogously, extend to CCS/PC the results about observational equivalence (say [7, Proposition 7.1, Theorems 7.2, 7.3 and 7.4]).

The overall result of Theorems 2.2 and 2.3 and similar could be stated in a more general form, as a metatheorem (like the 'duality principle' in projective geometry). We do not think that such formalization is worthwhile here, but simply notice that in general propositions about behaviour programs in CCS, whose proofs use only axioms and rules of the derivation system, hold also when the variables involved are considered ranging over CCS/PC behaviour expressions. That is because the form of behaviour programs and the derivation rules are the same in CCS and in CCS/PC.

This remark justifies our claim that CCS theory transfers almost unchanged to CCS/PC.

3. Examples of applications

Just for the sake of clarity, in the following examples it can be convenient to associate with every positive label constant α a fixed domain of values $D(\alpha)$, intuitively the 'type' of values allowed on α. In this case we make the following further assumptions on the syntax of behaviour expressions.

For every positive label expression Φ with free variables $x_1, \ldots, x_n$, if $\Phi\{v_1/x_1, \ldots, v_n/x_n\}$ evaluates to α and $\Phi\{v_1'/x_1, \ldots, v_n'/x_n\}$ evaluates to α', then $D(\alpha) = D(\alpha')$. Hence, we can define $D(\Phi)$ setting $D(\Phi) = D(\alpha)$, where $\Phi\{v_1/x_1, \ldots, v_n/x_n\}$ evaluates to α for some $v_1, \ldots, v_n$.

In $\Phi x_1, \ldots, x_n$ and $\bar{\Phi}E_1, \ldots, E_n$ the tuples $\tilde{x} = x_1, \ldots, x_n$ and $\tilde{E} = E_1, \ldots, E_n$ are bound to assume values in $D(\Phi)$.

Relabellings must respect types, i.e., $D(S(\alpha)) = D(\alpha)$ for every $S: \Lambda \to \Lambda$ and every $\alpha \in \Lambda$.

In the following, when not differently specified, we consider a simple syntax for positive label expressions: $\phi ::= \alpha | \alpha(\mathrm{expr})$, where expr ranges over a set of expressions with values in an enumerable set V_{expr} and if expr evaluates to $v \in V_{\mathrm{expr}}$, then $\alpha(\mathrm{expr})$ evaluates to a positive label α_v (in a family of distinct positive labels $\{\alpha_v \mid v \in V_{\mathrm{expr}}\} \subset \Delta$).

3.1. *A queue of processes*

We want to model a process which schedules access to a resource with a first come-first served policy. Let R denote the resource, U_i $(i \geq 0)$ the users, $Q(s)$ the scheduler, assuming that s ranges over strings of integers and initially $s = \Lambda$.

$R \Leftarrow \overline{R\text{-ready}}.\alpha_R.\cdots$ interaction with the served process $\ldots .R$,

$U_i = \bar{\alpha}_Q i.\mathrm{go}_i.\bar{\alpha}_R.\cdots$ interaction with the resource $(i \geqslant 0)$,

$Q(\Lambda) \Leftarrow \alpha_Q n.Q(n)$,

$Q(ms) \Leftarrow \alpha_Q n.Q(msn) + R\text{-ready}.\overline{\mathrm{go}(m)}.Q(s)$,

where m, n are variables with values in $\mathbb{N}$; s is a variable with values in $\mathbb{N}^*$; Λ denotes the empty string; R-ready, α_R, go_i $(i \geqslant 0) \in \Delta$.

The value sets associated to the label expressions are defined by $D(R\text{-ready}) = D(\alpha_R) = D(\mathrm{go}_i) = \emptyset$ and $D(\alpha_Q) = \mathbb{N}$.

This queue works as follows: Q can receive on α_Q a request from the ith user and in this case Q stores i in the queue; Q can also receive on R-ready from the resource R the signal that R is available, and in this case Q allows the use of R to the first process in the queue—m—by sending a message on $\overline{\mathrm{go}}_m$ and deletes m from the queue. Note that, as long as the queuing processes are taken from a finite set, the above model of a queue can be encoded in CCS, by a technique similar to the one shown in [7, p. 114].

Again, as remarked above, if the number of processes is infinite, that solution does not work. Moreover, it is easy to realize that, when it works, that encoding represents a very low level implementation of queue; in this respect, while it provides, via translation, a well-founded semantics, it obscures sensibly the abstract notion of queue.

3.2. Procedures in a concurrent language

As we already have said in the introduction, parametric channels allow the parallel execution of many simultaneous instances of a procedure. In order to give a better understanding of the technique, instead of considering one particular language, we present different solutions discussing under which assumptions they work; i.e., we adopt a stepwise explanatory approach (similar to Milner's style in [7, p. 133], but there a fixed language scheme is considered).

Let us go back to the discussion in the introduction, recalling the notations used there and still assuming, for the moment, that procedures do not call themselves. A first simple idea for eliminating the indicated drawbacks of the translation into CCS is modifying the translation of a procedure declaration as follows:

$$g \Leftarrow \alpha_G(N, x).(g \mid (\mathrm{LOC}_X \mid \mathrm{LOC}_Y \mid \bar{\alpha}_X x.[\![\mathrm{body}]\!] \text{ before } \gamma_Y y.\overline{\gamma - G(N)}y.\mathrm{NIL}) \setminus L_X \setminus L_Y),$$

assuming that N is a variable with values in a set of identifiers. The behaviour program g receives on α_G a call to the procedure G, which specifies, together with the value x to be assigned to the input parameter X, also a name N which identifies the call; the receiving of this message causes the creation of a new copy of g (for allowing further calls) and the execution of the body of the procedure; at the end g sends the result y on the correct line $\gamma - G_N$ (depending on N).

Correspondingly, we have to associate identifiers to calls. Since here only processes can call procedures, we could attribute to a call the name of the process in which a call takes place. This solution works only if we can ensure that the association of names to processes is such that contemporary calls do not use the same name; for example, if a process splits into two parallel processes, say by a forking operation, then we should attribute to them new different names. As for the association of a name to a call, the simplest case is when we have a finite number of processes, with names statically determined by the syntax; then the translation of a procedure call can be:

$$[\![G(\text{expr}, Z)]\!] = ([\![\text{expr}]\!] \mid \rho x.\bar{\alpha}_G(I, x).\gamma - G_I z.\bar{\alpha}_Z z.\bar{\varepsilon}.\text{NIL})\backslash\rho,$$

where I identifies the name of the process. Recall that in that case also a solution in CCS is possible, just using a corresponding number of copies of G. But note that then the translation of a procedure declaration depends on the number of processes.

When the association of names to processes and hence to procedure calls is dynamic, then a standard technique is to insert in parallel with each running process a register ID (for 'identification') storing the process name, initialized at the creation of a process with a new name taken from a name generating process. An example of this technique, though in a different context, will be given later in the following subsection. Then in translating a procedure call we have to provide for reading the process name from the register ID, e.g., as follows:

$$[\![G(\text{expr}, Z)]\!] = ([\![\text{expr}]\!] \mid \rho x.\gamma_{\text{ID}} N.\bar{\alpha}_G(N, x).\gamma - G(N) z.\bar{\alpha}_Z z.\bar{\varepsilon}.\text{NIL})\backslash\rho.$$

A suitable use of the restriction operator has to ensure that each process can only communicate with the right ID register.

A change in the translation of a procedure declaration is required, when allowing procedures to call themselves. Let us retain for the moment the technique using the identification register ID and the corresponding translation of a procedure call. A reasonable extension of the previous approach is that not only a direct procedure call in a process I should retain I as identifier, but also all the nested indirect calls. Here too the idea works, if we can avoid name confusion; for example, if procedure bodies do not involve processes, by associating names to processes, as before, in a way preventing contemporary calls to the same procedure by two processes having the same name. Then the following translation of a procedure declaration, which generalizes the technique for handling recursion in Milner [7, p. 135], guarantees the correct links. Assume that $G_1, \ldots, G_n$ are a n-tuple of procedures which can call each other and let g_i be the behaviour identifier corresponding to G_i,

$$g_i \Leftarrow \alpha_{G_i}(N, x).(g_1 \mid (\text{LOC}_X \mid \text{LOC}_Y \mid \bar{\alpha}_X x.([\![\text{body}]\!] \mid \text{ID}(N))\backslash\gamma_{\text{ID}} \mid$$
$$g_1 \mid \cdots \mid g_n)\backslash L_{G_1}\backslash \cdots \backslash L_{G_n} \text{ before } \gamma_Y y.\overline{\gamma_i - G(n)} y.\text{NIL})\backslash L_X \backslash L_Y),$$
$$\text{ID}(N) \Leftarrow \bar{\gamma}_{\text{ID}} N.\text{ID}(N),$$

where $L_{G_i} = \{\alpha_{G_i}, \gamma - G_i(I) \mid I \text{ process name}\}$.

The introduction of these inner copies of $g_1, \ldots, g_n$ allows to handle recursion. All the previous solutions, though nice and appealing in their use of parametric channels, need some extra condition preventing name confusion. But parametric channels do also permit a totally different approach, which is quite general and in this respect seems to encompass all the previous ones: the procedure itself assigns an order number to each call; in other words, every call uses a different copy of the procedure, as it is shown below.

$$[\![\textbf{proc } G(X:\textbf{in}\,;Y:\textbf{out})\text{body}]\!] = g(0), \text{ where}$$

$$g(n) \Leftarrow \bar{\eta}_G n.\alpha - G(n)x.(g(n+1) \mid (\mathrm{LOC}_X \mid \mathrm{LOC}_Y \mid \bar{\alpha}_X x.[\![\text{body}]\!] \text{ before } \gamma_Y y.\overline{\gamma - G(n)}y.\mathrm{NIL})\backslash L_X \backslash L_Y),$$

$$[\![G(\text{expr}, Z)]\!] = ([\![\text{expr}]\!] \mid \rho x.\eta_G n.\overline{\alpha - G(n)}x.\gamma - G(n)z.\bar{\alpha}_Z z.\bar{\varepsilon}.\mathrm{NIL})\backslash\rho.$$

In this case, a call of a procedure G consists first of all of a request on the line η_G, which g will answer by sending an order number n to the calling process. That process can use the nth copy of the procedure G through the line $\alpha - G(n)$ and $\gamma - G(n)$ while the $(n+1)$st copy is created for allowing further calls. Notice that the last approach strictly needs parametric channels. An analogous technique will be used in Section 4 in the translation of the same procedure declaration in SCCS.

With the above techniques we could, for example, complete our translation of Brinch Hansen's Distributed Processes given in [2], removing the assumption there made of no parallel executions of a procedure.

3.3. A language with process variables

We consider a language ML which is a fragment of the metalanguage used in [4] for describing the semantics of ADA. For the concurrent structure that metalanguage is based on an extension of CSP devised by Bjørner and Folkjaer [3]. In ML we have included those features, namely process instantiation and a communication scheme with process variables, which are of interest for illustrating the use of parametric channels. In the full metalanguage of [3] the communication mechanisms are more variate and complex, but their treatment does not require a substantially different technique. Apart from further exemplifying the use of parametric channels, the relevance of this translation should be clear: it means that the formal specification of ADA tasking given in [4] can be expressed in CCS/PC, and hence supported by a nice mathematical foundation. But note that the specification given in [4] does not correspond exactly to the informal semantics of ADA given in the reference manual as noticed by the authors themselves: for example, timed statements should have a different description, which is not possible in CCS/PC.

Syntax of ML:

program ::= processor list ; initialize,
processor list ::= processor | processor ; processor list,

processor ::= p: **processor** (var) = (decl list ; clause list),
p ::= identifier,
initialize list ::= initialize | initialize ; initialize list,
initialize ::= **initialize** p(expr),
decl list ::= decl | decl ; decl list,
decl ::= ivar : Π | var : type,
clause list ::= clause | clause ; clause list,
clause ::= nd-input clause | output clause | start clause | ...,
nd-input clause ::= (input clause list),
input clause list ::= input clause | input clause ; input clause list,
input clause ::= **input** var **from** ivar $\Rightarrow$ clause list,
output clause ::= **output** expr **to** ivar $\Rightarrow$ clause list,
start clause ::= [ivar ::=] **start** p(expr).

An ML program consists of a dynamically varying number of disjoint processes, each one identified by a name (here a natural number) and being an instance of a processor definition declared at the beginning of the program. The initialize part specifies the instances which start at the beginning of the execution; new instances can be created by executing start clauses. A start clause **start** p(expr) specifies that a new instance, based on the processor definition identified by p, starts running in parallel with the instances which already exist, having as actual parameter the value of expr. With this new instance is associated a new, unique name which identifies it. Every instance has its own variables; variables of type Π, ranged over by π, are used to store instance names.

Instances communicate via input/output clauses which are an extended version of CSP ones. More precisely, an output clause '**output** expr **to** $\pi \Rightarrow$ clause list' specifies as a partner, instead of a statically known process (as in CSP), the instance whose name is stored in π. (Analogously for input clauses.)

Translation. We write just $P_1|P_2|P_3$ instead of either $(P_1|P_2)|P_3$ or $P_1|(P_2|P_3)$ because they are strongly equivalent (analogously for +). We give a translation scheme following the order of the syntax clauses, with interleaved explanatory comments.

In what follows, n, m are variables with values in $\mathbb{N}$ and Pr ranges over ML programs.

$$[\![\text{Pr}]\!] = [\![\text{processor list ; initialize list}]\!]$$
$$= ([\![\text{processor list}]\!] | [\![\text{initialize list}]\!] | \text{NAMES}(0)) \backslash A_{\text{Pr}},$$

where

$$A_{\text{Pr}} = \{\text{who}_1, \text{who}_2, \text{start}_p, \text{from}\text{—}i\text{—}\text{to}\text{—}j \mid p \text{ is a processor definition identifier in Pr}, i, j \in \mathbb{N}\}.$$

$$\text{NAMES}(n) \Leftarrow \overline{\text{who}_1}\, n.\overline{\text{who}_2}\, n.\text{NAMES}(n+1),$$

where $\text{who}_1, \text{who}_2 \in \Delta$ and $D(\text{who}_1) = D(\text{who}_2) = \mathbb{N}$.

NAMES is just a generator of new instance names (identified here by natural numbers). It sends the name n of a new instance on the line $\overline{\text{who}_1}$ to the started instance itself, on the line $\overline{\text{who}_2}$ to the starter instance which can store it in an own variable of type Π.

In the following, PID denotes the set of processor definition identifiers, ranged over by p, and X denotes a variable.

$$[\![\text{processor};\text{processor list}]\!] = [\![\text{processor}]\!] \,|\, [\![\text{processor list}]\!],$$

$$[\![p\text{: }\textbf{processor}\ (X) = (\text{decl list};\text{clause list})]\!] = \omega(p),$$

where

$$\omega(\text{pvar}) \Leftarrow \text{start}(\text{pvar})x.\text{who}_1 n.(((\text{LOC}_X \,|\, \bar{\alpha}_X x.[\![\text{decl list};\ \text{clause list}]\!]) \backslash L_X \,|\, \text{ID}(n)) \backslash \gamma_{\text{ID}} \,|\, \omega(\text{pvar})),$$

$$\text{ID}(n) \Leftarrow \bar{\gamma}_{\text{ID}} n.\text{ID}(n),$$

where $\gamma_{\text{ID}}, \text{start}_p \in \Delta$; $D(\text{start}_p)$ is the set of values which can be stored in X and pvar is a variable with values in PID.

The translation of a processor definition p is a recursively defined program $\omega(p)$ which, after receiving a start signal on start_p, produces a new instance identified by a new name received on the line who_1 from NAMES. $\text{ID}(n)$ is just a 'read only' register which stores the 'identity' of an instance, in the sense that n is the integer which identifies the instance.

$$[\![\text{initialize};\text{initialize list}]\!] = [\![\text{initialize}]\!] \,|\, [\![\text{initialize list}]\!],$$

$$[\![\textbf{initialize}\ p(\text{expr})]\!] = ([\![\text{expr}]\!] \,|\, \rho x.\overline{\text{start}_p} x.\text{who}_2\, n.\text{NIL}) \backslash \rho.$$

The initialize part always activates the first instance of p, say with name n. Notice that the input guard $\text{who}_2\ n$ is only present for homogeneity with the translation of the start clauses (see later), so that NAMES(n) can pass to NAMES($n+1$).

$$[\![\text{decl list};\text{clause list}]\!] = ([\![\text{decl list}]\!] \,|\, [\![\text{clause list}]\!]) \backslash L_{\text{decl list}},$$

where $\backslash L_{\text{decl list}}$ is a shorter notation for $\backslash L_{X_1} \backslash \cdots \backslash L_{X_k}$ if $X_1, \ldots, X_k$ are the variables declared in 'decl list';

$$[\![\text{decl};\text{decl list}]\!] = [\![\text{decl}]\!] \,|\, [\![\text{decl list}]\!],$$

$$[\![X\text{: type}]\!] = \text{LOC}_X,$$

$$[\![\pi\text{: }\Pi]\!] = \text{LOC}_\pi, \quad \text{where } D(\alpha_\pi) = D(\gamma_\pi) = \mathbb{N};$$

$$[\![\text{clause};\text{clause list}]\!] = [\![\text{clause}]\!] \text{ before } [\![\text{clause list}]\!],$$

$$[\![(\text{input clause};\text{input clause list})]\!] = [\![\text{input clause}]\!] + [\![\text{input clause list}]\!],$$

$$[\![\textbf{input}\ X\ \textbf{from}\ \pi \Rightarrow \text{clause list}]\!] =$$

$$= \gamma_{\text{ID}} n.\gamma_\pi m.\text{from—to—}(m, n)x.\bar{\alpha}_X x.[\![\text{clause list}]\!],$$

$$[\![\textbf{output expr to } \pi \Rightarrow \text{clause list}]\!] =$$

$$= ([\![\text{expr}]\!] \mid \rho x.\gamma_{\text{ID}} n.\gamma_{\pi} m.\overline{\text{from—to—}(n, m)}x.[\![\text{clause list}]\!])\backslash\rho,$$

$$[\![[\pi :=] \textbf{ start } p(\text{expr})]\!] = ([\![\text{expr}]\!] \mid \rho x.\overline{\text{start}_p}\, x.\text{who}_2 n.[\bar{\alpha}_{\pi} n.]\bar{\varepsilon}.\text{NIL})\backslash\rho.$$

The label expression from—to—(i, j) evaluates to from—i—to—j ($\in \Delta$) for every $i, j \in \mathbb{N}$. Notice that the channel used in a communication depends on the name of the instance itself and on the instance name stored in the ivariable appearing in the clause. Since these two names are not known from the syntax, but the first is dynamically assigned at the moment of the creation and the second is determined at run time when reading the ivariable, nonconstant label expressions are needed here. Notice, moreover, in the translation of a start clause the optional part $\bar{\alpha}_{\pi} n$ (not appearing in the translation of the initialize), which allows the storage in π, for use in communication of the name n received from NAMES (if the "$\pi :=$" part is present).

4. Extensions and discussion

CCS/PC is an extension of CCS which increases the power of CCS, while not lowering the level of the language; the extension consists essentially of admitting more powerful expressions, leaving invariate the communication mechanism.

Another completely different approach for extending the expressive power of CCS has been taken by Milner (see [8]), who has devised a new calculus, SCCS, which is much more powerful and general than CCS. Though the comparison is more subtle than it appears at first sight, SCCS is in some sense a lower-level calculus than CCS, especially because it eliminates values and hence expressions; but values can be encoded in SCCS and, moreover, the full CCS and CCS/PC can be reobtained as subcalculi of SCCS (see [8, Part 2]). Clearly SCCS provides a very elegant and unifying foundational framework, which we consider a truly remarkable achievement.

We believe that both approaches are useful, in order to provide modularity in dealing with semantics of concurrency; in particular, they can be applied to give semantics by translation.

In this section we discuss, illustrate and implicitly compare the two approaches, first indicating possible extensions of CCS/PC and then showing by an example how the parametric channels problem can be solved in SCCS.

Extensions. In the preceding section we gave the translation of a fragment of the metalanguage used in [4]. In the other mentioned paper [3], Bjørner and Folkjaer admit a more flexible version of I/O clauses, like

ouput expr **to one in** π – set expr ; [**set** π ;],

where π – set expr is an expression whose evaluation produces a set of instance names. In this case the possible receivers are a set of instances which is evaluated

at run time. Label expressions as they have been introduced are no longer sufficient to describe this situation; a further extension of CCS is needed, where we replace label expressions Φ with label-set expressions Ψ which evaluate to sets of labels. Hence if $\Psi \rightarrow^* \alpha$, for α in a set of labels L, then $\Psi x.P \rightarrow^{\alpha} P\{v/x\}$ for α in L. Notice that this corresponds to removing the assumption of uniqueness in the evaluation of label expressions, and then the parametric channels described in this paper are only a special case of this more general description. Notice also that such an extension would correspond to a bigger (proper) subset of CTs than CCS/PC (for example, the CT of Fig. 2 would be expressible). Clearly, further extensions in the same spirit are possible; e.g., instead of label expressions we could consider label-value expressions which evaluate to sets of pairs (label, value), thus allowing one to express the tree in Fig. 3.

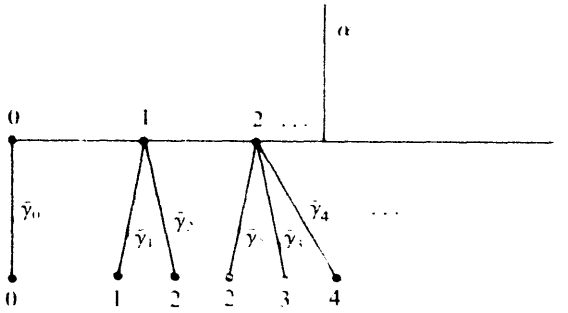

Fig. 3.

Finally, we mention other extensions that we have considered in some stage of our work (the first has been included in the Conference paper; the second has also been suggested by one of the referees):

(1) admitting parametric relabellings, i.e., considering behaviour expressions $B[R]$, where R is an expression which evaluates to a relabelling;

(2) allowing parametric restrictions, i.e., considering behaviour expressions $B\backslash\Phi$, where Φ denotes a label expression.

In the examples found so far we did not need such extensions, but we cannot exclude that they can become of value in some applications. As for the relative theory, from what we have seen so far, it seems plausible to anticipate that no essential changes would be needed, but that should be checked in detail.

Parametric channels in SCCS. SCCS, introduced by Milner [8], is a synchronous calculus without value passing and with only four operators and recursion without parameters, in which the binary operator + is replaced by an infinite indexed sum. Value and label passing/parametric channels can be suitably encoded in SCCS. We will give here not a formal encoding of CCS/PC into SCCS, but, as an example, a translation into SCCS of the procedure case mentioned in Section 1.2.

$[\![\textbf{proc}\ G(X:\textbf{in}\,;y:\textbf{out})\ \text{body}]\!] = g_{(\ }$

where

$$g_i \Leftarrow \delta\Big(\sum_{x\in\omega} \alpha - G_{i,x} : (g_{i+1} \times (\mathrm{LOC}_X \times \mathrm{LOC}_Y \times \overline{\alpha - X_x} : [\![\mathrm{body}]\!] \text{ before } \sum_{y\in\omega} \gamma - Y_y : \overline{\gamma - G_{i,y}} : \mathbb{1}) \backslash\backslash L_X \backslash\backslash L_Y)\Big),$$

$$[\![G(\mathrm{expr}, Z)]\!] = \Big([\![\mathrm{expr}]\!] \times \delta\Big(\sum_{x\in\omega} \rho_x : \sum_{i\in\omega} \overline{\alpha - G_{i,x}} : \delta\Big(\sum_{z\in\omega} \gamma - G_{i,z} : \overline{\alpha - Z_z} : \bar{\varepsilon} : \mathbb{1}\Big)\Big)\Big) \backslash\backslash \rho,$$

where

$$\mathrm{LOC}_X \Leftarrow \delta\Big(\sum_{m\in\omega} \alpha - X_m : \mathrm{REG} - X_m\Big),$$

$$\mathrm{REG} - X_n \Leftarrow \delta\Big(\sum_{m\in\omega} \alpha - X_m : \mathrm{REG} - X_m + \overline{\gamma - X_n} : \mathrm{REG} - X_n\Big),$$

$L_X = \{\alpha - X_n, \gamma - X_n \mid n \in \omega\}$ for any register X,

E before $E' = (E[\phi] \times \delta(\beta : E')) \backslash\backslash \beta$

(where $\phi = \beta/\varepsilon$, β new) for any SCCS expression E, E'.

The definition of a register shows one of the essential features of SCCS: values are coded into integer indexes. While referring to Milner [8] for a full explanation of SCCS, we give here the essential elements for understanding the example.

SCCS is a synchronous language, in the following sense: the composition operator of CCS, whose behaviour is described saying that in $P_1 | P_2$ either P_1 and P_2 synchronously communicate or may move independently (i.e., asynchronously) is replaced by a product $\times$, such that in $E_1 \times E_2$ (E ranges over SCCS expressions), both E_1 and E_2 must move together in a synchronous way. Hence when we want to allow an SCCS expression to wait, we must use the explicit delay operator δ. Thus, for example, the effect of prefixing SCCS expressions by $\delta\alpha$: operator is the same as prefixing a CCS expression by an α operator.

$\backslash\backslash$ is analogous to the restriction operator $\backslash$ in CCS; ϕ is a morphism, which is a suitable notion replacing CCS relabelling.

As we already did for the composition operator in CCS and CCS/PC, we use $\times$ as an n-ary, instead of binary, operator, because it is associative.

$\mathbb{1}$ can be viewed here by the reader just as a new version of NIL.

In the proposed translation, the behaviour of a procedure is modelled by an infinite family of procedure copies g_i, $i \in \omega$, using the fact that in SCCS we can write $\sum_{i\in I} E_i$, where I is an enumerable set of indexes. Thus a call to G will be answered by exactly one g_i and in this way the results of a procedure execution can be returned to the correct calling process without risk of intereferences (which was the problem

with CCS), because every calling process uses a particular procedure copy. The example above, as well as giving the flavour of SCCS, illustrates, in our opinion, when compared with the previous translation into CCS/PC, the advantage of a modular approach to translation: use flexible extensions of CCS for translating a concurrent language and encode the extensions into SCCS, as a foundational calculus.

Acknowledgment

We are grateful to all the referees, both of the Conference paper and of the present one, for their careful comments, which have been of great help in making the presentation more readable and accurate.

References

[1] E. Astesiano and E. Zucca, Semantics of CSP via translation into CCS, *Proc. 10th Symposium on MFCS*, Lecture Notes Comput. Sci. **118** (Springer, Berlin, 1981).

[2] E. Astesiano and E. Zucca, Semantics of distributed processes derived by translation, *Proc. 11th GI-Jahrestagung*, Informatik Fachberichte **50** (Springer, Berlin, 1981).

[3] D. Bjørner and P. Folkjaer, A formal model of a generalized CSP-like language, in: *Information Processing 80* (North-Holland, Amsterdam, 1980).

[4] D. Bjørner and O. Oest, Towards a formal description of ADA, Lecture Notes Comput. Sci. **90** (Springer, Berlin, 1980).

[5] C.A.R. Hoare, Communicating sequential processes, *Comm. ACM* **21** (8) (1978).

[6] M. Hennessy, W. Li and G. Plotkin, A first attempt at translating CSP into CCS, *Proc. 2nd International Conference on Distributed Computing Systems* (IEEE Comput. Soc. Press, MD, 1981).

[7] R. Milner, *A Calculus of Communicating Systems*, Lecture Notes Comput. Sci. **92** (Springer, Berlin, 1980).

[8] R. Milner, Calculi for synchrony and asynchrony, *Theoret. Comput. Sci.* **25** (3) (1983) 267–310.

[9] G. Plotkin, A structural approach to operational semantics, Lecture notes, DAIMI FN-19, Aarhus University, 1981.