Interpreting one Concurrent Calculus in Another

Citation for published version:
Milner, R 1990, 'Interpreting one Concurrent Calculus in Another' Theoretical Computer Science, vol. 75, no. 12, pp. 3-13. DOI: 10.1016/0304-3975(90)90059-Q

Digital Object Identifier (DOI):
10.1016/0304-3975(90)90059-Q

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Publisher's PDF, also known as Version of record

Published In:
Theoretical Computer Science

Theoretical Computer Science 75 (1990) 3-13
North-Holland

INTERPRETING ONE CONCURRENT CALCULUS IN ANOTHER

Robin MILNER

Department of Computer Science, Edinburgh University, King's Buildings, Mayfield Road, Edinburgh, UK EH9 3JZ

1. Introduction

It seems natural to use different languages for the different purposes of description or programming on the one hand, and prescription or specification on the other hand. Certainly there have been recent attempts to conflate these two types of language, but in the author's opinion they have not been convincing. It is, however, the purpose of this paper not to argue this point, but rather to explore a consequence of accepting the use of different languages for the two purposes.

I propose to use the word "description" as a generalisation of "program"; it is something which will describe both the spatial or modular structure of a performing agent (hardware or program) and also the temporal details of its performance. A prescription or specification, on the other hand, defines *properties* of the agent's performance—but only those properties which are expressible in terms of that part of the performance which is observable, e.g. the initial and final values of a program's memory, or the program's intermediate interactions with the user, or the electrical behaviour at the boundary of a chip. The connectives or operators for building specifications are naturally logical; many of those for building descriptions are not naturally logical but express operational or structural ideas like sequencing, interaction and juxtaposition.

If it is natural to use two languages, it is essential to define the relationship between them. It is not just that the designer (programmer, or describer) and the specifier should be able to interact, but that the two languages and what they express must form a single conceptual framework in which both designer and specifier operate (indeed, they may be the same person). We shall use the term *calculus* to mean a pair of languages in a definite relationship.

Having defined *calculus*, we then look at what it means to interpret one calculus in another. We have in mind that some calculi are primitive and general, but hard to apply in practice, while others are powerful and specific, designed for particular applications, and easier to work in. We therefore propose that it is useful to build an applied calculus on top of a basic one, i.e. interpret the former in the latter, to avoid an anarchic plethora of different calculi. The main result of the paper is that a particular applied calculus, namely a Hoare logic for an imperative concurrent programming language, may be interpreted in the author's Calculus of Communicating Systems [5, 7].

2. Calculi and their inter-relation

Definition 2.1. A *calculus* is a triple

$$\mathscr{C} = (\mathscr{A}, \models, \mathscr{L})$$

where

(1) $\mathscr{A}$ is term *algebra*: terms built by a given set of operators;
(2) $\mathscr{L}$ is a *logic*: formulae built by a given set of connectives;
(3) $\models \subseteq \mathscr{A} \times \mathscr{L}$ is the *satisfaction relation* between terms and formulae.

This is a very bare definition, and leaves many options open. For example the interpretation of terms $t \in \mathscr{A}$ may be given independently, or as an equivalence induced by $\models$ as follows:

$$t_1 \equiv t_2 \text{ iff for all formulae } F \in \mathscr{L}, \quad t_1 \models F \text{ iff } t_2 \models F.$$

Also, the satisfaction relation $\models$ may be presented in different ways; either in terms of the model theory of $\mathscr{A}$ and $\mathscr{L}$, or proof-theoretically as an inference system whose sentences are of the form $t \models F$. In the latter case we shall call $\mathscr{C}$ a *proof calculus*.

The notion of calculus is not so refined as the notion of *institution* in [1]; in particular, we are not concerned here with variation of signature in a calculus, nor with their condition which requires the satisfaction relation to be preserved under signature-change. But the motivation for calculi is somewhat the same as for institutions. Here, we are mainly concerned with a single example of the relationship between two calculi; in studying calculi more generally we may well wish to enter the framework of [1], and the possibility of doing so must be investigated.

Our motivation for calculi is as follows. For different design purposes (designing programs in different languages, or designing hardware) different descriptive and prescriptive languages, i.e. different calculi, will be appropriate; but one will often wish to make use of the properties, e.g. satisfactions, of one calculus when working in another. To this end, we have to set up relationships among calculi which will admit this transfer of properties between them.

Among many possible relationships, we wish to give an illustration of just two, with respect to specific calculi of interest in parallel computation.

Definition 2.2. Let $\mathscr{C} = (\mathscr{A}, \models, \mathscr{L})$ and $\mathscr{C}' = (\mathscr{A}', \models', \mathscr{L}')$ be two calculi. We say h *derives* C' from $\mathscr{C}$ if h is a pair (h_1, h_2) of functions, $h_1 : \mathscr{A}' \to \mathscr{A}$ and $h_2 : \mathscr{L}' \to \mathscr{L}$, and $t \models' F$ iff $h_1(t) \models h_2(F)$. We say $\mathscr{C}'$ *includes* $\mathscr{C}$ if $\mathscr{A} \subseteq \mathscr{A}'$, $\mathscr{L} \subseteq \mathscr{L}'$ and $\models \subseteq \models'$. In this case we write $\mathscr{C} \subseteq \mathscr{C}'$.

The way we shall use these ideas is as follows. We take a simple basic calculus of processes,

$$\mathrm{PC} = (\mathrm{PA}, \models, \mathrm{PL}).$$

We then take a simple but practical imperative programming language, presented as an algebra IA, whose operators are the syntactic constructions of the language, and define its semantics by a semantic function $\mathcal{M}_1 : \mathrm{IA} \to \mathrm{PA}$. A natural Hoare logic which goes with IA is then expressed as the calculus

$$\mathrm{IC} = (\mathrm{IA}, \vdash, \mathrm{IL})$$

where IL is essentially a set of pairs of formulae of predicate logic, and $\vdash$ expresses the inference rules of the Hoare logic. Thus IC is a proof calculus.

So far, we have the incomplete diagram

$$\begin{tikzcd}\mathrm{PA} \arrow[r, phantom, "\models"] & \mathrm{PL} \\ \mathrm{IA} \arrow[u, "\mathcal{M}_1"] \arrow[r, phantom, "\vdash"] & \mathrm{IL}\end{tikzcd}$$

We therefore seek a translation $\mathcal{M}_2 : \mathrm{IL} \to \mathrm{PL}$, to complete the diagram in a certain sense. Ignoring $\vdash$ for a moment, we shall then have a calculus IC′ derived by $(\mathcal{M}_1, \mathcal{M}_2)$ from PC, i.e.

$$\mathrm{IC}' = (\mathrm{IA}, \models', \mathrm{IL})$$

where if $C \in \mathrm{IA}$ and $F \in \mathrm{IL}$ then $C \models' F$ is defined to mean $\mathcal{M}_1(C) \models \mathcal{M}_2(F)$. Since F is a pair (P, Q), we think of $\mathcal{M}$ as an interpretation of the Hoare sentences $P\{\!|C|\!\}Q$ in PC.

Finally, we find that $\mathrm{IC} \subseteq \mathrm{IC}'$, i.e. $\vdash \subseteq \models'$. We propose that this is the correct way to formulate and to prove that a proof calculus is *sound* with respect to its interpretation in an underlying calculus.

3. The process calculus

Our *process calculus*

$$\mathrm{PC} = (\mathrm{PA}, \models, \mathrm{PL})$$

has been described at length elsewhere, and we shall only review it briefly here.

The *process algebra* PA consists of the terms of CCS in its pure form [6, 7]. We start with a set $\mathcal{N} = \{a, b, c, \ldots\}$ of names, $\bar{\mathcal{N}} = \{\bar{a}, \bar{b}, \bar{c}, \ldots\}$ of *co-names*, together forming the set $\mathcal{L} = \mathcal{N} \cup \bar{\mathcal{N}}$ of *labels*. These distinguish the *ports* at which communication occurs between agents; communication is always between a pair of ports with complementary labels, such as a and $\bar{a}$. We extend complementation to the whole of $\mathcal{L}$ by setting $\bar{\bar{a}} = a$. The labels also denote the *actions* occurring at ports. There is a special action τ; it is the action performed by a composite agent when two of its component agents, running concurrently, interact with each other by performing a pair of complementary actions. We let $\alpha, \beta, \ldots$ range over $Act = \mathscr{L} \cup \{\tau\}$, the set of *actions.*

For agents, we first introduce a set PK of *agent constants*; we let $A, B, \ldots$ range over PK. We let $P, Q, \ldots$ range over PA, the set of *agents*, given by the following syntactic rules:

$P ::= \alpha.P$	action prefix;
$\mid \Sigma_{i \in I} P_i$	summation (I an indexing set);
$\mid P_1 \mid P_2$	composition;
$\mid P \backslash L$	restriction ($L \subseteq \mathscr{L}$);
$\mid P[f]$	relabelling (where f is a partial function on $\mathscr{L}$ and $f(\bar{\alpha}) = \overline{f(\alpha)}$);
$\mid A$	constant.

We write $\mathbf{0}$ for the empty summation $\sum_{i \in \emptyset} P_i$.

We further require for each constant A a defining equation of the form

$$A \stackrel{\text{def}}{=} P.$$

For example, the definition

$$A \stackrel{\text{def}}{=} a.b.A + c.A$$

represents the agent which can repeatedly perform *either* the two actions a, b in sequence *or* the single action c. The behaviour of agents is defined as a labelled transition system with transition relations $\xrightarrow{\alpha}$ ($\alpha \in Act$), and the above agent is fully described by the transitions

$$A \xrightarrow{a} b.A, \quad A \xrightarrow{c} A, \quad b.A \xrightarrow{b} A.$$

Space precludes a full definition of these relations, but we shall treat one further example which we shall need later. We wish to model the behaviour of a storage register which may hold an arbitrary natural number. Thus the constant REG_n (for each $n \geqslant 0$) represents the register in the state in which n is stored. In this state it may either be *assigned* a new content m, by performing the action a_m, or it may deliver up its *content n* by performing the action $\bar{c}_n$. The defining equations of the agents REG_n therefore take the form

$$\text{REG}_n \stackrel{\text{def}}{=} \sum_{m \geqslant 0} a_m.\text{REG}_m + \bar{c}_n.\text{REG}_n.$$

Often we write such equations as a single parametric equation

$$\text{REG}(x) \stackrel{\text{def}}{=} a(y).\text{REG}(y) + \bar{c}(x).\text{REG}(x)$$

using the convention that a *positive* label, here a, *binds* a value variable while a negative label, here c, can be parameterised by a value expression. Note that REG_n

has the following transitions:

$$\mathrm{REG}_n \xrightarrow{a_m} \mathrm{REG}_m \quad \text{for each } m \geq 0,$$

$$\mathrm{REG}_n \xrightarrow{\overline{c_n}} \mathrm{REG}_n.$$

An agent P interacting with the register, on the other hand, will perform an action labelled $\bar{a}_m$ to assign a particular integer m to the register, and must be capable of performing any action c_n, $n \geq 0$, to read the contents of the register. If two agents P_1 and P_2 can both perform such actions (at various times in their histories) then the restricted composition

$$(P_1 \,|\, P_2 \,|\, \mathrm{REG}_0)\backslash\{a, c\}$$

represents the system in which they, and (because of the restriction) *only* they, can make use of the register, perhaps competing for its attention.

Despite its simplicity, PA is a convenient vehicle for the expression and analysis of non-trivial parallel systems which are of practical importance. Many examples can be found in the author's book [7].

Let us now turn to the component PL of the calculus PC; the *process logic*. The only basic material from which the *formulae* of PL are constructed is the set *Act* of actions. We let $F, G, \ldots$ range over PL, which is defined by the following syntactic rules:

$F ::= \langle\alpha\rangle F$	progression ($\alpha \in Act$);
$\mid \bigwedge_{i\in I} F_i$	conjunction (I an indexing set);
$\mid \neg F$	negation.

We write **true** for the empty conjunction $\bigwedge_{i\in\emptyset} F_i$. Intuitively, when we assert $\langle\alpha\rangle F$ of an agent P, we mean that P has a transition $P \xrightarrow{\alpha} P'$ such that F holds of the agent P'. Formally, the satisfaction relation $\models$ which completes our calculus PC is defined as follows, by induction on the structure of formulae:

(1) $P \models \langle\alpha\rangle F$ if, for some P', $P \xrightarrow{\alpha} P'$ and $P' \models F$;

(2) $P \models \bigwedge_{i\in I} F_i$ if, for all $i \in I$, $P \models F_i$;

(3) $P \models \neg F$ if it is not the case that $P \models F$.

Now PL, like PA, is a simple, but also powerful. With a few derived connectives, a wide range of properties of behaviour can be expressed very succinctly; for example, the ability or inability of a process to reach a deadlocked state is easy to express. Here are a few derived forms:

$$\langle s\rangle \text{ means } \langle\alpha_1\rangle\langle\alpha_2\rangle\ldots\langle\alpha_n\rangle F \quad (s = \alpha_1 \ldots \alpha_n)$$

$$[s]F \text{ means } \neg\langle s\rangle\neg F$$

$$\bigvee_{i\in I} F \text{ means } \neg\bigwedge_{i\in I}\neg F$$

As an example, let us suppose that we wish to express the property of REG (above) so that, whatever actions it performs, it can always yield some natural number as its content. Then the following formula F expresses the property, where *Nat* is the set of natural numbers,

$$F \equiv \bigwedge_{s\in Act^*} [s] \bigvee_{m\in Nat} \langle c_m\rangle\textbf{true}.$$

Moreover, it is easy to establish that, for each $n \in Nat$,

$$\text{REG}_n \models F.$$

Now PL can be used to induce congruences upon the term algebra PA, and by dividing PA by any such congruence one obtains an interesting and rich algebraic theory. The simplest congruence is known as *strong congruence* and denoted by $\sim$; it is defined according to the general recipe mentioned directly after the definition of *calculus* in Section 2, namely

$$P \sim Q \quad \text{if for all } F \in \text{PL}, \quad P \models F \text{ iff } Q \models F.$$

The algebraic properties of this congruence, and others, can be found in [7]. It is worth noting that these congruences have other characterisations independent of PL, and this gives them more objective status. We need not pursue this matter further here; we have now treated PC enough for our present purpose, namely the interpretation in PC of the imperative calculus IC, to which we now turn.

4. The imperative calculus

As announced earlier, our imperative calculus

$$\text{IC} = (\text{IA}, \rightharpoondown, \text{IL})$$

is, in essence, the Hoare logic of a simple imperative programming language. As basic sets we take $\mathscr{X}$, the program variables, and $\mathscr{F}$, the function symbols (each function symbol having an arity $\geqslant 0$). We let X and F range over $\mathscr{X}$ and $\mathscr{F}$, respectively. We also let E range over the *expressions* $\mathscr{E}$, and C and range over the *commands* $\mathscr{C}$, defined by the following syntactic rules:

$E ::= X$ variable	
$\mid F(E_1, \ldots, E_n)$	function application
$C ::= X := E$	assignment
$\mid C; C'$	sequential composition
$\mid$ **if** E **then** C **else** C'	conditional
while E **do** C	iteration
local X **in** C **end**	local variable
C **par** C'	parallel composition
skip	no action

(Note that constants like 0, 1, ... and arithmetic operations like +, −, ... are function symbols with arity zero and arity two respectively, and we would write $X+1$ in place of $+(X, 1(\ ))$.)

Thus the *imperative algebra* IA is a term algebra with two sorts (expressions and commands) and with the syntactic constructions as operators. These constructions are well enough known to need little description. Suffice to say that the local variable construction gives scope C to the variable X, and that in the parallel composition construction C and C' are supposed to run in parallel, communicating through variables to which they both share access.

Now Hoare's original logic [3] for partial correctness of sequential programs employed sentences of the form

$$P\{C\}Q$$

whose intended meaning is "if C is executed starting in a state satisfying P, then its terminating state (if any) will satisfy Q". P and Q are normally taken to be formulae of first order logic, containing free occurrences of the program variables. A natural rule of inference is then the following rule for sequential imposition:

$$\frac{P\{C_1\}Q \quad Q\{C_2\}R}{P\{C_1;C_2\}R}.$$

In fact, as is well known, Hoare and others have given sets of inference rules for various sequential languages. If we were only concerned with sequential programs, without **par**, then we would define the *imperative logic* IL to have pairs (P, Q) as its formulae (where P, Q are formulae of predicate logic), and we would take $P\{C\}Q$ to be a way of writing $C \vdash (P, Q)$. Then we would complete the imperative proof calculus IC by defining the relation $\vdash$ as the set of pairs $(C, (P, Q))$ such that $P\{C\}Q$ is provable in the appropriate Hoare logic.

In the presence of **par** things are not so easy, because without some constraint upon the language there is no natural inference rule corresponding to **par**. However, as Owicki and Gries showed [8], there *is* a natural rule if we impose the following condition: In any command of the form C_1 **par** C_2, C_1 may not assign to any program variable which occurs free in C_2, and conversely. We shall proceed to formulate IL and $\vdash$ with this in mind.

First, we shall decorate each Hoare sentence with two disjoint sets $\tilde{X}$ and $\tilde{Y}$ of program variables, as follows:

$$P\{C\}Q|^{\tilde{X}}_{\tilde{Y}}$$

We shall require that $\tilde{Y}$ contains all the variables free in C to which C makes any assignment, and $\tilde{X}$ contains all other variables free in C; also, that $\tilde{X} \cup \tilde{Y}$ contains all the program variables free in P and Q. If these conditions are all satisfied then we say the sentence is *admissible*. Now the rule for sequential compositions becomes the following:

$$\frac{P\{C_1\}|^{\tilde{X}_1}_{\tilde{Y}_1} \quad Q\{C_2\}R|^{\tilde{X}_2}_{\tilde{Y}_2}}{P\{C_1;C_2\}R|^{(\tilde{X}_1-\tilde{Y}_2)\cup(\tilde{X}_2-\tilde{Y}_1)}_{\tilde{Y}_1\cup\tilde{Y}_2}}$$

where it may easily be checked that the conclusion is admissible provided that the hypotheses are so. For parallel composition, we impose a side condition on

$\tilde{X}_1$, $\tilde{X}_2$, $\tilde{Y}_1$ and $\tilde{Y}_2$ which ensures the above stated condition on C_1 and C_2, assuming that the hypotheses are admissible:

$$\frac{P_1\{\!|C_1|\!\}Q_1|^{\tilde{X}_1}_{\tilde{Y}_1} \quad P_2\{\!|C_2|\!\}Q_2|^{\tilde{X}_2}_{\tilde{Y}_2}}{(P_1 \wedge P_2)\{\!|C_1 \textbf{ par } C_2|\!\}(Q_1 \wedge Q_2)|^{\tilde{X}_1 \cup \tilde{X}_2}_{\tilde{Y}_1 \cup \tilde{Y}_2}}$$

provided that $\tilde{X}_1 \cap \tilde{Y}_2 = \tilde{Y}_1 \cap \tilde{X}_2 = \tilde{Y}_1 \cap \tilde{Y}_2 = \emptyset$. Note that under these conditions the conclusion will also be admissible. It is rather easy to complete the Hoare logic by supplying inference rules for the other constructions of IA, and we shall take them for granted.

We therefore modify our definition of IL, the *imperative logic*; it consists of quadruples $(P, Q, \tilde{X}, \tilde{Y})$ and we take $P\{\!|C|\!\}Q|^{\tilde{X}}_{\tilde{Y}}$ to be a way of writing $C \Vdash (P, Q, \tilde{X}, \tilde{Y})$. Then we complete IC by defining $\vdash$ as the set of pairs $(C, (P, Q, \tilde{X}, \tilde{Y}))$ such that $P\{\!|C|\!\}Q|^{\tilde{X}}_{\tilde{Y}}$ is admissible, and provable in the Hoare logic.

Much has been written about the soundness and (relative) completeness of various Hoare logics. We do not address the problem of completeness in this paper; however, the soundness of the imperative calculus is the subject of the following section.

5. Interpreting the imperative calculus

Let us review what we mean by the soundness of IC with respect to its interpretation in PC. First we have to express the interpretation as a pair of translations

$$\mathcal{M}_1 : \text{IA} \to \text{PA}, \quad \mathcal{M}_2 : \text{IL} \to \text{PL}.$$

Then we have to show that, for any $C \in \text{IA}$ and any $(P, Q, \tilde{X}, \tilde{Y}) \in \text{PA}$,

$$C \vdash (P, Q, \tilde{X}, \tilde{Y}) \text{ implies } \mathcal{M}_1(C) \models \mathcal{M}_2(P, Q, \tilde{X}, \tilde{Y}).$$

In this section, we outline the translations $\mathcal{M}_1$ and $\mathcal{M}_2$, and also outline the proof of soundness.

The translation $\mathcal{M}_1$ from imperative programs to CCS agents rests upon the simple idea that each program variable X corresponds to a relabelled version of the register agent defined in Section 3 above, i.e.

$$\text{REG}_X(x) \stackrel{\text{def}}{=} a_X(y).\,\text{REG}_X(y) + \bar{c}_X(x).\text{REG}_X(x).$$

Moreover, the translation $\mathcal{M}_1(C)$ of any command C, in which the program variable X occurs free, will be an agent capable of performing actions $\bar{a}_X$ (to assign to X) and actions c_X (to obtain the contents of X). It is particularly important to realise that no assumption is made, in defining $\mathcal{M}_1$, that a program C will have *exclusive* access to the variables which it uses. Thus, for example, the translation $\mathcal{M}_1(\mathbf{Y} := \mathbf{X} + \mathbf{X})$ will have many possible execution sequences; it can perform the sequence

$$\bar{c}_{\mathbf{X}}(6)\bar{c}_{\mathbf{X}}(6)a_{\mathbf{Y}}(12)$$

which is what it will do if the variable $\mathbf{X}$ contains the value 6 unchanged throughout the execution of **Y**:=**X**+**X**; but alternatively it can perform the sequence

$$\bar{c}_X(6)\bar{c}_X(7)a_Y(13)$$

which is what it will do if some other agent increments the value of **X** between the two readings of that variable.

The full translation is given in [5, 7]; here we need only give its flavour by indicating how it treats local variables. The local variable construction

$$C' = \textbf{local } X \textbf{ in } C \textbf{ end}$$

has two important effects. First, it dedicates the variable X to C, ensuring that no other agent can access X; second, it ensures that any $\bar{a}_X$ or c_X actions which occur in the behaviour of C (representing the writing and reading of X by C) are replaced by τ actions in the behaviour of C'. These are both achieved by the simple definition

$$\mathcal{M}_1(\textbf{local } X \textbf{ in } C \textbf{ end}) = (\text{REG}_X(0)\,|\,\mathcal{M}_1(C))\backslash\{a_X, c_X\}$$

(where we assume X is initially given the value 0).

The remainder of the definition of $\mathcal{M}_1$ is not hard, but need not concern us here. It has the pleasant property that in the theory of PA, divided by one of the congruences mentioned at the end of Section 3, one can easily prove many familiar equational laws for the transformation of programs. Indeed, a principle motive for investigating $\mathcal{M}_1$ was to obtain those laws as a justification for the algebraic theory PA.

We now turn to defining $\mathcal{M}_2$: IL $\longrightarrow$ PL. That is, we look for a uniform construction of a modal formula F from a quadruple P, Q, $\tilde{X}$, $\tilde{Y}$ such that $C \models F$ asserts, of C, the following: Let m, and m' be two valuations of the variables $\tilde{X} \cup \tilde{Y}$, and let s be a terminating action sequence of C which can lead from initial valuation m to final valuation m' assuming no external interference with the variables $\tilde{X} \cup \tilde{Y}$. Then if m satisfies P, m' satisfies Q.

Now let us write $\tilde{Z} = \tilde{X} \cup \tilde{Y}$, and let us write Dom$(m)$ for the set of variables for which the valuation m is defined. Furthermore, let us denote by $m\{\!|s|\!\}m'$ the property that the s is a terminating action sequence, and could consistently lead from initial valuation m to final valuation m' without external interference (this property is quite easy to define inductively on s). Let us write $P\{m(\tilde{Z})/\tilde{Z}\}$ for the formula obtained by replacing in P the variables $\tilde{Z}$ by their values in m. Then the formula $\mathcal{M}_2(P, Q, \tilde{X}, \tilde{Y})$ may be written as a mixture of predicate and modal logic as follows:

$$\begin{aligned} \forall m, m', s.\quad & (\text{Dom}(m) = \text{Dom}(m') = \tilde{Z} \wedge m\{\!|s|\!\}m' \\ & \wedge \langle s\rangle\textbf{true} \wedge P\{m(\tilde{Z})/\tilde{Z}\}) \\ & \supset Q\{m'(\tilde{Z})/\tilde{Z}\}. \end{aligned}$$

Now we first assert that this formula can be encoded entirely in PL. This may seem surprising, but the power of infinite conjunction in PL, with arbitrary indexing sets, is what makes it possible. Second, we assert that if $P\{\!|C|\!\}Q|^{\tilde{X}}_{\tilde{Y}}$ is interpreted as

$\mathcal{M}_1(C) \models \mathcal{M}_2(P, Q, \tilde{X}, \tilde{Y})$ then the rule for parallel composition in Section 4, and indeed all the other rules of the Hoare logic described there, are sound. (In this proof, the side condition on the rule for **par** is indeed crucial.) Therefore everything that is provable by these rules is true in the interpretation in PC, and we have demonstrated the soundness of the imperative calculus with respect to the underlying process calculus, as we intended.

6. Conclusion

In this paper we have shown how one calculus can be interpreted in another, more basic, calculus. The particular example we chose was to interpret an imperative programming calculus (likely to be more familiar to applications specialists) in the basic process calculus CCS. This interpretation was rather easy and natural, and lends weight to CCS as a foundation for calculi which are more oriented to particular applications; one hopes that CCS can similarly support many applied calculi.

Indeed there is a calculus, based upon the same imperative programming language IA, which presents an immediate challenge of the above kind. For our proof calculus IC is quite restrictive; the domain of its satisfaction relation $\vdash$ is confined to programs in which the parallel construction C_1 **par** C_2 is only admitted when C_1 assigns to no variable occurring free in C_2 in vice versa. This condition ensures that programs are deterministic, but it is an unnecessarily strong condition to impose for this purpose. There are interesting deterministic programs in IA which do not satisfy the condition; indeed, there are useful non-deterministic programs which are intuitively natural and which one would like to analyse in a richer calculus than IC.

Such richer calculi exist, for the same imperative programming language IA. One good example stems from the work of Jones [4]; in effect, he replaces Hoare sentences $P\{\!\{C\}\!\}Q$ by richer sentences $(P, R)\{\!\{C\}\!\}(Q, S)$, where P and Q are pre- and post-conditions as before, while R is a condition upon whose truth C will *rely* before every step of its execution, and S is a condition whose truth C will, in turn, *guarantee* after every step of its execution. Now Jones places no restriction upon the parallel construction C_1 **par** C_2, but will only allow a Hoare sentence about C_1 **par** C_2 to be inferred from Hoare sentences about C_1 and C_2 whose *rely* and *guarantee* conditions complement one another suitably. Stirling [9] has formulated the appropriate Hoare logic explicitly, as a generalisation of the Owicki-Gries system, and has proved it sound with respect to an independently given operational semantics. To justify PC as a good foundation, it is therefore important to formulate Stirling's Hoare logic as a calculus in our sense, and interpret it in PC just as we have interpreted IC in PC.

Hitherto we have concentrated upon Hoare logics of *partial* correctness: logics which assert nothing about program termination. There are logics of *total* correctness for the same programming language IA, for example the weakest pre-condition logic of Dijkstra. It appears that our process logic PL does not express *liveness* or *eventuality* properties strongly enough for PC to support an interpretation of a logic of total correctness. However, there are stronger process logics than PL which do express eventuality, Hennessy and Stirling [2] proposed one, and we would like to interpret a total correctness calculus for IA in a correspondingly stronger process calculus.

We hope to have shown that the notions of calculus, and of interpreting one calculus in another, are fruitful and unifying.

Note added in proof

Since this paper was written, Tofts has succeeded in interpreting Stirling's richer calculus in PC in his PhD thesis [10].

References

[1] J.A. Goguen and R.M. Burstall, Introducing institutions, in: E. Clarke and D. Kozen, eds., *Proc. Logics of Programming Workshop*, Lecture Notes in Computer Science **164** (Springer, Berlin, 1984) 221-256.

[2] M.C. Hennessy and C.P. Stirling, The power of the future perfect in program logics, *Inform. and Control* **67** (1985) 23-52.

[3] C.A.R. Hoare, An axiomatic basis for computer programming, *Comm. ACM* **21**(8) (1969) 576-580.

[4] C.B. Jones, Specification and design of (parallel) programs, in: *Proc. IFIP 9th World Computer Congress* (North-Holland, Amsterdam, 1983) 321-332.

[5] A.J.R.G. Milner, *A Calculus of Communicating Systems*, Lecture Notes in Computer Science **92** (Springer, Berlin, 1980); also available as Report ECS-LFCS-86-7, Computer Science Department, University of Edinburgh, 1986.

[6] A.J.R.G. Milner, Calculi for synchrony and asynchrony, *Theoret. Comput. Sci.* **25** (1983) 267-310.

[7] A.J.R.G. Milner, *Communication and Concurrency* (Prentice Hall, Englewood Cliffs, NJ, 1989).

[8] S. Owicki and D. Gries, An axiomatic proof technique for parallel programs I, *Acta Inform.* **6**(1) (1976) 319-340.

[9] C.P. Stirling, A generalisation of Owicki-Gries' Hoare logic for a concurrent while language, *Theoret. Comput. Sci.*, to appear.

[10] C. Tofts, Proof methods and pragmatics for parallel programming, PhD thesis, Computer Science Department, University of Edinburgh, 1990.