Theoretical Computer Science 26 (1983) 105–120
North-Holland

A MORE COMPLETE MODEL OF COMMUNICATING PROCESSES

E.C.R. HEHNER
Computer Systems Research Group, University of Toronto, Toronto M5S 1A1, Canada

C.A.R. HOARE
Oxford University Computing Laboratory, Programming Research Group, Oxford OX2 6PE, United Kingdom

Communicated by M. Nivat
Received January 1982
Revised October 1982

Abstract. A previous paper by Hoare gives axioms and proof rules for communicating processes that provide a calculus of total correctness. This paper gives explicit definitions of communicating processes as predicates. The former axioms and proof rules become theorems, proved using the explicit definitions. The defining predicates are more powerful than the proof rules for reasoning about processes, but less often useful for their construction. An implementation of the processes using partial recursive functions is given.

Introduction

Axioms and proof rules for communicating processes have been given by Hoare [1]. They serve three purposes:

(a) they implicitly define communicating processes;

(b) they can be used to reason about communicating processes;

(c) they can be used to construct communicating processes for a given purpose.

In this paper, processes are given explicit definitions as predicates. The axioms and proof rules in [1] become theorems about processes, proved using the explicit definitions in this paper. This proves the soundness of the calculus presented in [1].

The proof rules are incomplete as a means of reasoning about communicating processes; an example of an expressible but not provable truth is given in [1]. The 'honest toil' of giving explicit predicate definitions has the advantage that it is complete, relative to data types. Its disadvantage is that the explicit definitions are less often useful for the construction of processes to fulfil a given purpose than the former proof rules (now theorems).

An axiomatic definition requires a model to demonstrate the consistency of the axioms. In [2], a set-theoretic model of communicating processes is given. From that, we know that at least one mathematical object satisfies the axioms. We should like to know that at least one computable mathematical object satisfies the axioms. To that end, an implementation using partial recursive functions is presented in this paper.

1. Axiomatic basis

A message is a pair ⟨channel, value⟩ over an alphabet of channels and values. If direction of communication is important, a message can be taken instead to be a triple ⟨channel, direction, value⟩, or channels can be considered directed; however, we shall not need to refer to the direction of communication. (This is entirely realistic; if you tap a phone line, you can record 'values', but not direction.) To denote a message, we shall use infix '!' instead of angle brackets. For example $c!1$ is a message with channel c and value 1.

A particular process at a particular moment has a past, which is the sequence of messages it has communicated up to that moment. It also has a present, which is the set of messages that it can communicate at the next step. Which message in the set it actually does communicate next may be determined in part (or completely, or not at all) by the environment of the process (those other processes connected to it by channels). If the set is empty, the process cannot communicate further, and is said to be deadlocked.

Let *past* be a free variable representing a finite sequence of messages. Let *present* be a free variable representing a finite set of messages. A predicate R in *past* and *present* is said to describe a process P if, at all times (before and after each communication) during any evolution (execution) of P, R is true of P's past and present. This is denoted

$$P\ \mathbf{sat}\ R$$

meaning "P is described by R", or "P satisfies description R".

Hoare proposes four 'healthiness conditions' that are reasonably required of the **sat** relation. For all processes P and all predicates R, S, $R(n)$,

(H1) $P\ \mathbf{sat}\ \mathbf{true}$.

(H2) $\neg(P\ \mathbf{sat}\ \mathbf{false})$.

(H3) If $R \Rightarrow S$ is a theorem, then $(P\ \mathbf{sat}\ R) \Rightarrow (P\ \mathbf{sat}\ S)$ is a theorem.

(H4) $(\forall n \in N.\ P\ \mathbf{sat}\ R(n)) \equiv (P\ \mathbf{sat}\ \forall n \in N.\ R(n))$.

On this basis, an axiom or proof rule is introduced in [1] for each construct (an axiom is just a proof rule with no premises).

For example, the process STOP can be defined by the axiom

$$(\text{STOP}\ \mathbf{sat}\ R) \equiv R[\mathit{past}: \langle\,\rangle; \mathit{present}: \{\,\}].$$

The right side denotes the predicate obtained from R by simultaneously substituting, for all free occurrences of *past*, the empty sequence of messages, and for all free occurrences of *present*, the empty set of messages.

2. Explicit basis

"P **sat** R" means that predicate R describes process P, but the description may be a weak one. In the extreme, according to (H1), **true** is a (weak) description of every process. By (H4), the conjunction of all descriptions of a process P is also a description of P; it is the strongest, and therefore an exact, description of P. If that description can be written explicitly, then it is an explicit definition of P.

Notationally, there is no reason to distinguish between a process and the predicate that defines it. Therefore we shall define a process as a predicate in the free variables *past* and *present* over a finite alphabet of messages. (The finiteness of the alphabet will be required later for computability.) For example, STOP can be defined as

$$\text{STOP}\colon \mathit{past} = \langle\,\rangle \wedge \mathit{present} = \{\,\}.$$

The colon means "is defined as". (It is appropriate to use the same symbol for definition and substitution because definition is simply permission to substitute.)

For any process P, the possible initial communications are described exactly by the predicate $P[\mathit{past}\colon \langle\,\rangle]$, and the sequences of communications that lead to deadlock are described exactly by the predicate $P[\mathit{present}\colon \{\,\}]$. If we wish to prove that a process cannot deadlock, we must prove

$$\forall \mathit{past}.\ \neg P[\mathit{present}\colon \{\,\}].$$

With this way of defining processes, "P **sat** R" means

$$\forall \mathit{past}, \mathit{present}.\ P \Rightarrow R.$$

The healthiness conditions degenerate nicely.

(H1) $\forall \mathit{past}, \mathit{present}.\ P \Rightarrow \mathbf{true}$ is a tautology.
(H2) $\neg(\forall \mathit{past}, \mathit{present}.\ P \Rightarrow \mathbf{false})$ asserts that P is satisfiable.
(H3) If $R \Rightarrow S$ is a theorem, then $(\forall \mathit{past}, \mathit{present}.\ P \Rightarrow R) \Rightarrow (\forall \mathit{past}, \mathit{present}.\ P \Rightarrow S)$ is a theorem. This is provable in the predicate calculus.
(H4) $(\forall n \in N.\ \forall \mathit{past}, \mathit{present}.\ P \Rightarrow R(n)) \equiv (\forall \mathit{past}, \mathit{present}.\ P \Rightarrow \forall n \in N.\ R(n))$ is a tautology, assuming P does not mention n.

The word 'healthy' has been reduced to the word 'satisfiable', and indeed a predicate should be true of some *past* and *present* if it is to be a reasonable definition of a process. That excludes

$$\text{MIRACLE}\colon \mathbf{false}$$

but is not sufficient as a characterization of processes. Consider

$$\text{NOWAY1}\colon \mathit{past} = \langle c\,!\,1\rangle \wedge \mathit{present} = \{\,\}.$$

Although NOWAY1 is clearly satisfiable, execution cannot begin, because NOWAY1$[\mathit{past}\colon \langle\,\rangle]$ is not satisfiable; there is no initial message. It is reasonable to require of a process that when something cannot happen, it will not happen. A different problem arises in

NOWAY2: $past = \langle\,\rangle \wedge present = \{c\,!\,1\}$.

Initially, the message $c\,!\,1$ can be communicated, but *past* remains forever empty. It is reasonable to require that when something can happen, something will happen.

Definition. A process P is a predicate in the free variables *past* and *present* such that

(P0) $\exists past, present.\ P$,

(P1) $\forall past, message.\ (\exists present.\ P \wedge message \in present)$

$$\equiv (\exists present.\ P[past\colon past\hat{\ }\langle message\rangle])$$

where ^ is catenation of message sequences.

Perhaps (P1) is best explained as two implications, slightly rearranged.

$$\forall past, present, message.$$
$$P \wedge message \in present \Rightarrow$$
$$\exists newpresent.\ P[past\colon past\hat{\ }\langle message\rangle;\ present\colon newpresent]$$

says that *past* can be extended by any *message* in *present*, and P will still be satisfiable (it still describes the process).

$$\forall past, present, message.$$
$$P[past\colon past\hat{\ }\langle message\rangle] \Rightarrow$$
$$\exists oldpresent.\ P[present\colon oldpresent] \wedge message \in oldpresent$$

says that a nonempty *past* can be shortened by removing its final *message*, and P must have been satisfied by that shorter *past* and a *present* containing that *message*.

According to the definition of a process, neither MIRACLE nor NOWAY1 nor NOWAY2 is a process. But STOP is, and so is

CHAOS: **true**.

The next section will show why this process merits the name "CHAOS".

Having proposed a definition of a process which we named STOP, we must prove that the definition of STOP is consistent with the axiom proposed for STOP in [1]. The axiom must become a theorem in the predicate calculus. To prove it, we need the following.

Substitution fact. *If P is a predicate, e is an expression, and x is a variable not appearing in e, then*

$$P[x\colon e] \equiv \forall x.\ x = e \Rightarrow P$$

where the quantifier ranges over the type of x.

Proof. In [3], Shoenfield proves

$$P[x:e] \equiv \exists x.\, x=e \wedge P$$

from which we know both

$$\neg(P[x:e]) \equiv \neg\exists x.\, x=e \wedge P,$$

$$(\neg P)[x:e] \equiv \exists x.\, x=e \wedge \neg P.$$

But

$$(\neg P)[x:e] \equiv \neg(P[x:e]).$$

Hence

$$\begin{aligned} P[x:e] &\equiv \neg\neg P[x:e] \\ &\equiv \neg\exists x.\, x=e \wedge \neg P \\ &\equiv \forall x.\, x=e \Rightarrow P. \quad \square \end{aligned}$$

STOP Theorem. (STOP **sat** R) $\equiv R[\mathit{past}: \langle\,\rangle; \mathit{present}: \{\}]$.

Proof. Using first the interpretation of "P **sat** R", then the definition of "STOP", then the substitution fact generalized to two variables, we have

$$\begin{aligned} \text{STOP } \mathbf{sat}\ R &\equiv \forall \mathit{past}, \mathit{present}.\ \text{STOP} \Rightarrow R \\ &\equiv \forall \mathit{past}, \mathit{present}.\ \mathit{past} = \langle\,\rangle \wedge \mathit{present} = \{\} \Rightarrow R \\ &\equiv R[\mathit{past}: \langle\,\rangle; \mathit{present}: \{\}]. \end{aligned}$$

3. Implementation

An implementation of a process is a partial recursive function from sequences of messages to sets of messages. For each past within its domain, the function determines (computes) a present that contains the possible next messages. If all of the processes connected by a channel are simultaneously willing to communicate several different messages, i.e. the intersection of their presents contains more than one message, then an arbitrary one of those messages will be communicated. In that way, the system of processes (not the individual processes) is nondeterministic.

There is another kind of nondeterminism. A process predicate P can be satisfied by a particular past and present, say s and M, and also by s and N; the same past can have more than one corresponding present. This process nondeterminism is a freedom for the implementor. An implementation, being deterministic, will deliver one of the corresponding presents for each past. In effect, for a function f to be an implementation of process P, it must be an implementation of some deterministic subprocess Q.

Definition. The pasts of a process P are defined as

$$\text{pasts}(P)\colon \{past \mid \exists present.\ P\}.$$

Definition. Q is a subprocess of P if Q is a process ((P0) and (P1) are true of Q) and $Q \Rightarrow P$.

Definition. Process P is deterministic if $\forall past \in \text{pasts}(P).\ \exists 1 present.\ P$.

Definition. An implementation of process P is a partial recursive function f from sequences to sets of messages, such that for some deterministic subprocess Q,

$$\forall s \in \text{pasts}(Q).\ Q[past\colon s;\ present\colon f(s)].$$

We shall use $\not\{P\not\}$ as the name of an implementation of process P. For example,

$$\not\{\text{STOP}\not\}(s)\colon \textbf{if } s = \langle\,\rangle \textbf{ then } \{\,\}$$

is a partial recursive function defined only for the one sequence $\langle\,\rangle$. It is a correct implementation of STOP because

(a) $\text{pasts}(\text{STOP}) = \{\langle\,\rangle\}$,

(b) $\text{STOP}[past\colon \langle\,\rangle;\ present\colon \not\{\text{STOP}\not\}(\langle\,\rangle)]$

$$\equiv \langle\,\rangle = \langle\,\rangle \wedge \{\,\} = \{\,\}$$

$$\equiv \textbf{true}.$$

All other correct implementations of STOP must agree with this one for the sequence $\langle\,\rangle$.

CHAOS is the most nondeterministic process. Every deterministic process is a subprocess of CHAOS, so that an implementation of any process is also an implementation of CHAOS. The name "CHAOS" is deserved because it is not at all determined what CHAOS will do.

4. Process construction

Processes can be constructed from other processes in prescribed ways. Suppose that F is a one-place process constructor, i.e. from process P we construct process $F(P)$, from Q, $F(Q)$, etc. We require F to obey these laws.

(PC0) If P is a process, then $F(P)$ is a process.

(PC1) (*monotonicity*): $\forall P, Q$. If $P \Rightarrow Q$ is a theorem, then $F(P) \Rightarrow F(Q)$ is a theorem.

(PC2) (*continuity*): If $P_0P_1P_2\ldots$ is a strengthening chain of processes, i.e. if $\forall i.\ P_{i+1} \Rightarrow P_i$ is a theorem, then $(\forall i.\ F(P_i)) \equiv F(\forall i.\ P_i)$ is a theorem.

By the usual trick known as 'Currying', a multi-place constructor can be regarded as the composition of several one-place constructors.

Each of Sections 5 to 14 presents one way of constructing processes from other processes. Each constructor is defined, a theorem (formerly axiom) is stated, and an implementation of it is displayed. For each constructor, it should be proved that (PC0)–(PC2) are satisfied, the theorem should be proved, and the correctness of the implementation should be proved. Most of these proofs are either tedious or obvious, and are omitted; only three of the more interesting proofs of theorems are included.

In addition to the standard notations of sets, predicates, and functions, the following notations will be used:

value(m)	is the value component of message m,
channel(m)	is the channel component of message m,
$\#s$	is the length of message sequence s,
$r\hat{}s$	is the catenation of message sequences r and s,
first(s)	is the first message in nonempty message sequence s,
rest(s)	is the subsequence of nonempty message sequence s that excludes the first message,
$s.c$	is the subsequence of message sequence s that includes those messages having channel component c,
$s\backslash c$	is the subsequence of message sequence s that excludes those messages having channel component c,
$M.c$	is the subset of message set M that includes those messages having channel component c,
$M\backslash c$	is the subset of message set M that excludes those messages having channel component c.

5. Output

A process that communicates the value of expression e on channel c (i.e. it communicates the message $c!e$) and then behaves like process P can be defined as

$$c!e \rightarrow P\colon \mathit{past} = \langle\,\rangle \wedge \mathit{present} = \{c!e\}$$
$$\vee\, \mathit{past} \neq \langle\,\rangle \wedge \mathrm{first}(\mathit{past}) = c!e \wedge P[\mathit{past}\colon \mathrm{rest}(\mathit{past})].$$

The output theorem, which will now be proved, is

$$(c!e \rightarrow P)\ \mathbf{sat}\ R \;\equiv\; R[\mathit{past}\colon \langle\,\rangle; \mathit{present}\colon \{c!e\}]$$
$$\wedge\, P\ \mathbf{sat}\ R[\mathit{past}\colon \langle c!e\rangle \frown \mathit{past}].$$

The proof has three parts.

(a) $(\forall \mathit{past}, \mathit{present}.\ \mathit{past} = \langle\,\rangle \wedge \mathit{present} = \{c!e\} \Rightarrow R)$
$\equiv\; R[\mathit{past}\colon \langle\,\rangle; \mathit{present}\colon \{c!e\}]$

by the substitution fact of Section 2.

(b) The predicate

$$\forall past, present.\ past \neq \langle\,\rangle \wedge \mathrm{first}(past) = c!e \wedge P[past: \mathrm{rest}(past)] \Rightarrow R$$

can be simplified by considering the two cases $past = \langle c!e\rangle\hat{\ }r$ for some sequence r, and $past \neq \langle c!e\rangle\hat{\ }r$ for any sequence r. In the latter case, the antecedent is clearly false, so the implication is clearly true. The implication must hold for all values of past, so it reduces to the former case,

$$\forall r, present.\ P[past: r] \Rightarrow R[past: \langle c!e\rangle\hat{\ }r].$$

This, by a change of bound variable, is just

$$\forall past, present.\ P \Rightarrow R[past: \langle c!e\rangle\hat{\ }past]$$

which proves

$$(\forall past, present.\ past \neq \langle\,\rangle \wedge \mathrm{first}(past) = c!e \wedge P[past: \mathrm{rest}(past)] \Rightarrow R)$$
$$\equiv (\forall past, present.\ P \Rightarrow R[past: \langle c!e\rangle\hat{\ }past]).$$

(c) Part (a) has proved something of the form

$$(\forall p.\ A \Rightarrow R) \equiv B.$$

Part (b) has proved something of the form

$$(\forall p.\ C \Rightarrow R) \equiv D.$$

From these two, follows

$$(\forall p.\ A \vee C \Rightarrow R) \equiv B \wedge D$$

which is the desired result.

An implementation of output is

```
$\nleftarrow c!e \to P \nrightarrow (s)$: **if** $s = \langle\,\rangle$
        **then** $\{c!e\}$
        **else if** $\mathrm{first}(s) = c!e$
            **then** $\nleftarrow P \nrightarrow(\mathrm{rest}(s))$.
```

6. Input

Let c be a channel, M a finite set of messages for channel c, and $P(x)$ a process with parameter x. Then $c?x:M \to P(x)$ is a process that can initially communicate any message in M, say $c!x$, and then behave like $P(x)$. It is defined as

$$c?x:M \to P(x):\ past = \langle\,\rangle \wedge present = M$$
$$\vee\ past \neq \langle\,\rangle \wedge \mathrm{first}(past) \in M$$
$$\wedge\ P(\mathrm{value}(\mathrm{first}(past)))[past: \mathrm{rest}(past)].$$

The input theorem

$$(c?x{:}M \rightarrow P(x))\ \textbf{sat}\ R$$

$$\equiv R\ [\mathit{past}{:}\langle\rangle;\mathit{present}{:}M]$$

$$\wedge\ \forall m \in M.\ P(\mathrm{value}(m))\ \textbf{sat}\ R[\mathit{past}{:}\langle m\rangle\hat{\ }\mathit{past}]$$

can be proved in a manner similar to the proof of the output theorem. In fact, according to their semantics, output is just the special case of input in which M is a unitset.

An implementation of input is

$$\{\!\!|c?x{:}M \rightarrow P(x)|\!\!\}(s){:}\ \textbf{if}\ s=\langle\rangle$$

$$\textbf{then}\ M$$

$$\textbf{else if}\ \mathrm{first}(s)\in M$$

$$\textbf{then}\ \{\!\!|P(\mathrm{value}(\mathrm{first}(s)))|\!\!\}(\mathrm{rest}(s)).$$

7. Recursion

If F is a one-place process constructor, then the recursive process definition

$$P{:}\ F(P)$$

means

$$P{:}\ \forall i.\ F^i(\mathrm{CHAOS})$$

where

$$F^0(P)=P, \qquad F^{i+1}(P)=F(F^i(P)).$$

Starting with CHAOS, we create an ordered sequence of processes $F^i(\mathrm{CHAOS})$. The ordering is reverse implication ('is implied by'). The sequence is one of ever more deterministic subproceses, and P is its limit. In other words, P is the least fixed point of F, where 'least' means 'weakest' or 'least deterministic'.

The theorem for recursively defined P is

$$(\forall i.\ (F^i(\mathrm{CHAOS})\ \textbf{sat}\ R\uparrow i) \Rightarrow (F^{i+1}(\mathrm{CHAOS})\ \textbf{sat}\ R\uparrow i+1)) \Rightarrow P\ \textbf{sat}\ R$$

where $R\uparrow i \equiv (\#\mathit{past}<i \Rightarrow R)$. To prove it, notice first that $R\uparrow 0 \equiv \textbf{true}$, hence $F^0(\mathrm{CHAOS})\ \textbf{sat}\ R\uparrow 0$. With that basis, the major antecedent, by induction, implies

$$\forall i.\ F^i(\mathrm{CHAOS})\ \textbf{sat}\ R\uparrow i$$

$$\forall i.\ \forall \mathit{past}, \mathit{present}.\ F^i(\mathrm{CHAOS}) \Rightarrow R\uparrow i$$

$$\forall \mathit{past}, \mathit{present}.\ \forall i.\ F^i(\mathrm{CHAOS}) \Rightarrow R\uparrow i$$

$$\forall \mathit{past}, \mathit{present}.\ (\forall i.\ F^i(\mathrm{CHAOS})) \Rightarrow (\forall i.\ R\uparrow i)$$

$$\forall \mathit{past}, \mathit{present}.\ P \Rightarrow R$$

$$P\ \textbf{sat}\ R$$

By choosing $\nless\mathrm{CHAOS}\ngtr(s)$ to be the universe of messages for the channels of the construction, P can be implemented as

$$\nless P\ngtr(s)\colon \bigcap_{i=0}^{\infty} \nless F^{i}(\mathrm{CHAOS})\ngtr(s)$$

assuming we know how to implement $F(P)$ in terms of an implementation of P. By choosing $\nless\mathrm{CHAOS}\ngtr(s)$ to be the empty set, P can be implemented as

$$\nless P\ngtr(s)\colon \bigcup_{i=0}^{\infty} \nless F^{i}(\mathrm{CHAOS})\ngtr(s).$$

8. Channel renaming

If process P does not mention channel d, then

$$P[c:d]$$

denotes a process like P except that all occurrences of channel c are replaced by channel d. Its theorem,

$$P[c:d]\ \mathbf{sat}\ R[c:d] \equiv P\ \mathbf{sat}\ R$$

where R does not mention c, is obvious. So is its implementation

$$\nless P[c:d]\ngtr(s)\colon \nless P\ngtr[c:d](s).$$

9. Disjoint parallelism

Let P be a process that communicates on the set of channels C, and let Q be a process that communicates on the set of channels D, where $C \cap D = \{\}$. Then $P|||Q$ is a process that behaves like P and Q in parallel, not communicating with each other. It is defined as

$$\begin{aligned} P|||Q\colon & P[\mathit{past}\colon \mathit{past.C};\ \mathit{present}\colon \mathit{present.C}] \\ & \wedge Q[\mathit{past}\colon \mathit{past.D};\ \mathit{present}\colon \mathit{present.D}] \end{aligned}$$

where the dot notation has been extended in an obvious way.

From this definition, the theorem

$$(P\ \mathbf{sat}\ S) \wedge (Q\ \mathbf{sat}\ T) \Rightarrow P|||Q\ \mathbf{sat}\ (S \wedge T)$$

is easily proved.

Disjoint parallelism can be implemented as

$$\nless P|||Q\ngtr(s)\colon \nless P\ngtr(s.C) \cup \nless Q\ngtr(s.D).$$

10. Channel connection

The process formed from process P by connecting its channels c and d and naming the connected channel b (a new channel name) is denoted

$$b = c \leftrightarrow d \textbf{ in } P.$$

The new process can communicate a message on the new channel b iff P can communicate a similar message on both channels c and d. For message $b!x$ to occur in the past of the new process, there must be a corresponding past of P in which $c!x$ and $d!x$ occur together. In fact, there must be two corresponding possible pasts of P, one in which $c!x$ and $d!x$ occur together in that order, and one in which they occur together in the other order. Without loss of generality, we can restrict our attention to one of the two orders. The definition can be stated as

$$\begin{aligned} b = c \leftrightarrow d \textbf{ in } P\text{: } & \exists s, M.\ P[past\text{: } s;\ present\text{: } M] \\ & \wedge \exists t.\ \forall i, x.\ (t_i = b!x) \equiv (s_i = c!x) \equiv (s_{i+1} = d!x) \\ & \wedge t \backslash b = s \backslash c \wedge past = t \backslash d \\ & \wedge present = \{b!x \mid \exists N.\ P[past\text{: } s\verb|^|\langle c!x\rangle\verb|^|\langle d!x\rangle;\ present\text{: } N]\} \\ & \qquad \cup M \backslash c \backslash d \end{aligned}$$

where t_i and s_i are the ith messages in message sequences t and s.

Its theorem is

$$\begin{aligned} (P \textbf{ sat } R) \Rightarrow (b = c \leftrightarrow d \textbf{ in } P) \textbf{ sat } & \exists x, y.\ R[c\text{: } x;\ d\text{: } y] \\ & \wedge present.b = present.x \cap present.y \\ & \wedge past.b = past.x = past.y. \end{aligned}$$

An implementation is

$$\nleftarrow b = c \leftrightarrow d \textbf{ in } P \nrightarrow (s)\text{: } \nleftarrow P \nrightarrow (g(s, \langle\,\rangle))$$

where

```
$g(s, t)$: if $s = \langle\,\rangle$
        then $t$
        else if first $(s) = b!x$
            then $g(\text{rest}(s), t\verb|^|\langle c!x\rangle\verb|^|\langle d!x\rangle)$
            else $g(\text{rest}(s), t\verb|^|\langle \text{first}(s)\rangle)$
```

reconstructs the past with $c!x$ and $d!x$ replacing each occurrence of $b!x$.

11. Hiding

If P is a process that communicates on channel b (and possibly others) then

chan b **in** P

is a process like P, except that channel b is local, and not visible to (available for communication with) its environment. Channel b is used for internal communication in the new process. Hiding is quite a delicate subject, as the discussion will show, and we do not yet have an entirely satisfactory definition. A first attempt is

chan b **in** P:

$$\exists s, M.\ P[past: s; present: M] \wedge past = s\backslash b \wedge present = M\backslash b.$$

A variation is obtained by adding the conjunct

$$\forall r, t, N.\ s = r\hat{\ }t \wedge P[past: r; present: N] \wedge N.b \neq \{\}$$
$$\Rightarrow t \neq \langle\,\rangle \wedge \text{channel}(\text{first}(t)) = b$$

within the scope of $\exists s$. Without this conjunct, a process is interruptible, i.e. there is the possibility of external communication as an alternative to internal communication. But with this conjunct, a process is uninterruptible while engaged in internal communication; when, in process P, communication is possible on either channel b or another channel, then, in process **chan** b **in** P, channel b will be chosen. The interruptible version includes the presents of unstable states, excluding the internal communications (hidden messages) that make the states unstable. It does so in order to include the possibility of external communications (interrupts) at those states. Unfortunately, this means that an implementation can deliver the present of an unstable state, ignoring forever the internal communication, waiting forever for an interrupt. The uninterruptible version includes only the presents of stable states. It is unsatisfactory because it excludes the possibility of external communication at an unstable state.

The hiding theorem is

$$(P\ \textbf{sat}\ (R \wedge \exists f.\ \#past.b \leq f(past\backslash b)))$$
$$\Rightarrow ((\textbf{chan}\ b\ \textbf{in}\ P)\ \textbf{sat}\ (\exists s.\ R[past: s] \wedge past = s\backslash b \wedge present.b = \{\}))$$

and can be proved from the uninterruptible version without using the theorem's antecedent. The purpose of the antecedent is to allow a definition of channel hiding that is weaker than the one we have given in the following way. In our uninterruptible version, a process which can engage in an infinite sequence of communications on the hidden channel does not communicate further with its environment; livelock (infinite internal chatter) is equivalent to deadlock. The hiding theorem, as an axiom in [1], allows livelock to be considered differently, in any way desired, even as equivalent to CHAOS; it does not allow anything to be proven about a livelocked process.

Hiding can be implemented, uninterruptibly, as follows.

$$⦃\textbf{chan } b \textbf{ in } P⦄(s)\colon ⦃P⦄(h(s, \langle\,\rangle))$$

where

```
$h(s, t)$: if $⦃P⦄(t).b = \{\}$
        then if $s = \langle\,\rangle$
            then $t$
            else $h(\text{rest}(s), t^\frown\langle\text{first}(s)\rangle)$
        else $h(s, t^\frown\langle\text{choose}(⦃P⦄(t).b)\rangle)$.
```

The function h constructs a new sequence t from the given sequence s such that $s = t\backslash b$ by inserting messages on channel b whenever possible. When there is a choice of messages that can be inserted, the choose function chooses an arbitrary one of them. Because *present* is always a finite set of messages, this choice will be from a finite set, and choose is therefore implementable. The function h is not necessarily terminating; this is an accord with the possibility of livelock.

The two kinds of nondeterminism mentioned in Section 3 can be called 'external' and 'internal'. External nondeterminism, having more than one message in a present, is resolved by the availability of external communication, and if that does not fully resolve it, then by other external forces. Internal nondeterminism, having more than one present, is resolved internally according to the implementation. But the two become mixed during channel hiding, when things external become internal. Separating them again properly is a delicate operation.

12. Nondeterministic union

The process that can behave either like process P or like process Q is defined as

$$P \textbf{ or } Q\colon P \vee Q.$$

Its theorem

$$(P \textbf{ or } Q) \textbf{ sat } R \;\equiv\; (P \textbf{ sat } R) \wedge (Q \textbf{ sat } R)$$

is easily proven.

One implementation is

$$⦃P \textbf{ or } Q⦄(s)\colon ⦃P⦄(s).$$

13. Conditional

The conditional process is defined as

$$\textbf{if } e \textbf{ then } P \textbf{ else } Q\colon e \Rightarrow P \wedge \neg e \Rightarrow Q$$

where e is a boolean expression, and P and Q are processes. Its theorem is

$$(\mathbf{if}\ e\ \mathbf{then}\ P\ \mathbf{else}\ Q)\ \mathbf{sat}\ R$$
$$\equiv\ e \Rightarrow (P\ \mathbf{sat}\ R) \wedge \neg e \Rightarrow (Q\ \mathbf{sat}\ R).$$

It can be implemented as

$$\nleftarrow\mathbf{if}\ e\ \mathbf{then}\ P\ \mathbf{else}\ Q\nrightarrow(s)\text{: } \mathbf{if}\ e\ \mathbf{then}\ \nleftarrow P\nrightarrow(s)$$
$$\mathbf{else}\ \nleftarrow Q\nrightarrow(s).$$

14. Alternation

Let I and J be input processes, as in Section 6. Then $I \,[\!]\, J$ is a process that behaves either like I or like J, depending on the availability of input for I and J. If, in its environment, an initial communication for I is possible but for J impossible, then it behaves like I. Conversely, if an initial communication for J is possible and for I impossible, it behaves like J. If both are possible, it behaves like either I or J. If neither is possible, it is deadlocked.

$$c?x:M \to P(x) \,[\!]\, d?y:N \to Q(y)\text{:}$$
$$\mathit{past} = \langle\,\rangle \wedge \mathit{present} = M \cup N$$
$$\vee\ \mathit{past} \neq \langle\,\rangle \wedge \text{first}(\mathit{past}) \in M$$
$$\wedge\ P(\text{value}(\text{first}(\mathit{past})))[\mathit{past}\text{: rest}(\mathit{past})]$$
$$\vee\ \mathit{past} \neq \langle\,\rangle \wedge \text{first}(\mathit{past}) \in N$$
$$\wedge\ Q(\text{value}(\text{first}(\mathit{past})))[\mathit{past}\text{: rest}(\mathit{past})].$$

The alternation theorem is

$$(c?x:M \to P(x) \,[\!]\, d?y:N \to Q(y))\ \mathbf{sat}\ R$$
$$\equiv R[\mathit{past}\text{: }\langle\,\rangle;\ \mathit{present}\text{: } M \cup N]$$
$$\wedge\ (\forall m \in M.\ P(\text{value}(m))\ \mathbf{sat}\ R[\mathit{past}\text{: }\langle m\rangle\hat{}\,\mathit{past}])$$
$$\wedge\ (\forall m \in N.\ P(\text{value}(m))\ \mathbf{sat}\ Q[\mathit{past}\text{: }\langle m\rangle\hat{}\,\mathit{past}]).$$

An implementation of alternation is

```
$\nleftarrow c?x:M \to P(x) \,[\!]\, d?y:N \to Q(y)\nrightarrow(s)$:
  if $s = \langle\,\rangle$
  then $M \cup N$
  else if $\text{first}(s) \in M$
```

```
        then ⦃P(value(first(s)))⦄(rest(s))
        else if first(s) ∈ N
            then ⦃Q(value(first(s)))⦄(rest(s)).
```

15. Conclusion

To program in the notation of communicating processes, one must begin with a specification, which is a predicate to be satisfied, and then choose an appropriate construction. An axiom or proof rule should say what the components of the construction must satisfy in order that the construction will satisfy the original predicate. Thus one creates specifications for subcomponents in the usual 'top-down' fashion.

The intended use of an object should guide the design of the object. Just as a specification, via the proof rules, guides the design of a process, so the desire for such proof rules guides the design of the range of process constructs. It is therefore appropriate for an axiomatic, implicit definition of communicating processes to precede an explicit definition. The explicit definition serves both as a guide for implementors, and as a more powerful tool for reasoning about processes. For example, defining

$$P : b!0 \to P$$

the predicate

$$(\textbf{chan } b \textbf{ in } P) \textbf{ sat true}$$

is cited in [1] as an expressible but unprovable truth. With the definitions in this paper, its proof is easy.

When processes are defined axiomatically, a model should be built to show that the axioms describe at least one object of interest; that was first done in [2]. The partial recursive functions in this paper serve as a second model, and a demonstration of computability. With more models, one gains confidence that at least some of the objects one wanted to define are in the class of objects defined.

How does one prove that the class defined includes all and only the desired objects? A Turing Machine simulation can show that all computable functions are expressible as processes, but the aptness of the process constructions is not amenable to proof in the usual sense. However, when a second, independent definition proves to be consistent with the first, confidence in the definitions is greatly increased. That is a contribution of this paper. In one sense, the predicate definitions are not independent: they were intended to be consistent with the axiomatic definitions. Nonetheless, they provide the reader with another look at communicating processes.

Acknowledgment

This paper was improved by the helpful comments of the referees, particularly Dr. Ernst-Rüdiger Olderog.

References

[1] C.A.R. Hoare, A calculus of total correctness for communicating processes, *Sci. Comput. Programming* **1**(1) (1981) 49–72.

[2] C.A.R. Hoare, S.D. Brookes and A.W. Roscoe, A theory of communicating sequential processes, Oxford University Programming Research Group Technical Monograph PRG-16 (1981).

[3] J.R. Shoenfield, *Mathematical Logic* (Addison-Wesley, Reading, MA, 1967) 36.