Towards a Theory of

COMMUNICATING SEQUENTIAL PROCESSES

C. A. R. HOARE

February 1979.

Summary. This paper extends the methods of a previous paper [ ] to describe nondeterministic processes. These are modelled as sets of deterministic processes. The problem of concealment of internal communication is solved. Some additional operators are defined, and their use illustrated in the design of some simple modules of an operating system.

Oxford University Computing Laboratory
Programming Research Group
45 Banbury Road,
Oxford OX2 6PE
England U.K.

1. Introduction.

A previous paper [ ] described a mathematical model of a communicating sequential process. Consider two such processes, P and Q, with the same alphabet. We define a nondeterministic process (P or Q) as a process which behaves either like P or like Q; but we cannot control (or even know in advance) which of the two it will be. Such a process can be modelled as the set $\{P,Q\}$. More generally, any nonempty set S of processes (with the same alphabet) defines a nondeterministic process, which is guaranteed to behave like some unknown and arbitrarily selected member of the set. Its alphabet $\Sigma(S)$ is the same as that of all its members.

The potential behaviour of S on its first step is denoted by S^o. This is a collection of sets of symbols which are acceptable on its first step by some member of S.

$$S^o =_{df} \{P^o \mid P \in S\}$$

If some symbol s in the union of S^o is accepted by S on its first step, then S(s) denotes the subsequent behaviour of S. It is defined

$$S(s) =_{df} \{P(s) \mid s \in P^o \ \& \ P \in S\}$$

This is empty if $s \tilde{\in} \bigcup S^o$, which indicates that s cannot be accepted by S on its first step.

Like a deterministic process, a nondeterministic process can be defined by an analogue of functional notation. Let Σ be an arbitrary alphabet; let C be an arbitrary collection of subsets of Σ; and let F be a function from $\bigcup C$ to nondeterministic processes with alphabet Σ.
We define

$$(s{:}C \rightarrow F(s)) =_{df} \{P \mid P^o \in C \ \& \ \forall s{:} \bigcup C \rightarrow P(s) \in F(s)\}$$

Clearly $(s{:}C \rightarrow F(s))^o = C$
and $(s{:}C \rightarrow F(s))(r) = F(r)$ for r in $\bigcup C$.

A nondeterministic process can be pictured as a tree, in which the existence of nondeterminism is indicated by a node with unlabelled arcs leading to the alternatives. Thus if

$$C = \{\{a\},\{a,b\}\}$$

then the process:

$$(s{:}C \rightarrow \underline{\text{if}}\ s = b\ \underline{\text{then}}\ \{(b \rightarrow \checkmark),(a \rightarrow \bot)\}\ \underline{\text{else}}\ \{b \rightarrow \bot\})$$

is shown in Fig 1. Regarded as a set, it contains:

$(a \to (b \to \bot))$
$(a \to (b \to \bot)) \sqcup (b \to (b \to \checkmark))$
$(a \to (b \to \bot)) \sqcup (b \to (a \to \checkmark))$

An alternative method of defining a process Q is by specifying the set of environments in which it may be successfully used. An environment P for Q is specified as a finite deterministic process, which is run in parallel with Q $(P||Q)$. The interaction is successful if the trace of the communications between Q and P is finite and ends in $\checkmark$. The interaction is certain to be successful if all traces of $Q||P$ are finite and end in $\checkmark$. Since we are interested only in the certainty of success, we define the specification of a process as the set of environments in which its use is certain to be successful:

$$\text{spec }(Q) =_{df} \{P \mid \Sigma(P) = \Sigma(Q) \,\&\, P \text{ is finite} \,\&\, \text{safe }(P||Q)\}$$

where safe $(T) =_{df}$ all branches of T end in $\checkmark$

This definition extends readily to a nondeterministic process S, whose specification is the set of environments in which it is certain to be successfully used:

$$\text{spec }(S) = \bigcap_{Q \in S} \text{spec }(Q)$$

Conversely, let T be an arbitrary set of environments. The implementation of T is the set of machines Q which can be successfully used in every environment of T

$$\text{imp }(T) = \bigcap_{P \in T} \{Q \mid P \in \text{spec }(Q)\}$$

If this set is empty, T is said to be an inconsistent specification, and the machine which satisfies it does not exist. Conversely, if T is empty then every machine (trivially) satisfies the specification T, even the machine which fails in every environment.

Consider now two nondeterministic processes with the same specification. For all practical purposes, these processes are equivalent to each other. In particular, each process is equivalent to the implementation of its own specification. So we may, without loss of generality, confine our attention to a canonical member of each equivalence class, which satisfies the equation

$$S = \text{can }(S)$$

where can $(S) = \text{imp }(\text{spec}(S))$.

Technical Note.

In order to justify the use of recursion in defining nondeterministic processes, we require a chain-complete partial ordering. A simple ordering, recommended by Smyth, is the superset ordering

$$S \sqsubseteq T =_{df} T \subseteq S$$

This definition reflects the fact that increased nondeterminism makes a process worse, because it is less predictable and less controllable. The worst possible process is one that is wholly arbitrary, and may behave in any way whatsoever.

The fact that this ordering is complete depends on finitude of the tests.

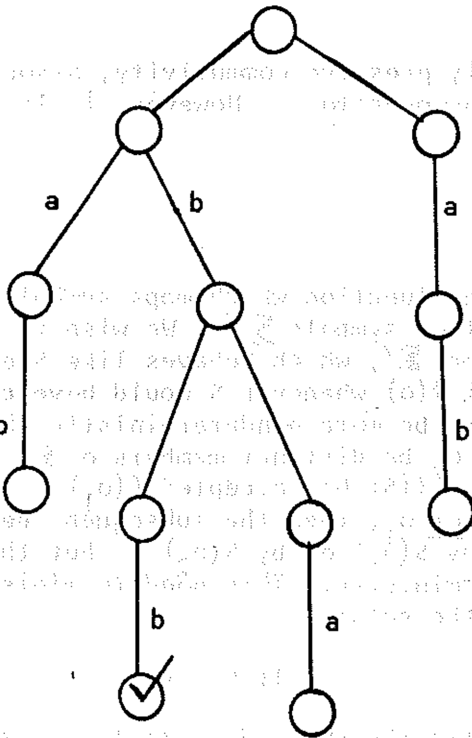

Fig 1.

2. Localisation of process names.

In this section we revise the model of communicating sequential processes given in [ ], and show a solution to the problem of localising process names.

2.1. Operators.

The operators defined in [ ] for single deterministic processes can be readily redefined for sets, by means of the usual pointwise extension; the result must then be made canonical:

$$
\begin{aligned}
S;T &= \text{can } \{(P;Q) \mid P \in S \,\&\, Q \in T\} \\
S \| T &= \text{can } \{(P \| Q) \mid P \in S \,\&\, Q \in T\} \\
a.S &= \text{can } \{(a.P) \mid P \in S\} \\
S \sqcup T &= \text{can } \{(P \sqcup Q) \mid P \in S \,\&\, Q \in T\}
\end{aligned}
$$

These definitions clearly preserve commutivity, associativity, and distributivity of the operators. However, $\|$ is no longer idempotent.

2.2. Symbol Change.

Let f be a many-one function which maps symbols of an alphabet $\Sigma(S)$ onto a set of symbols Σ'. We wish to define a process f(S) with alphabet Σ', which behaves like S except that it communicates a symbol $f(\sigma)$ whenever S would have communicated σ. In general, f(S) will be more nondeterministic than S. For example, let σ_1 and σ_2 be distinct members of Σ, such that $f(\sigma_1) = f(\sigma_2)$; and suppose f(S) has accepted $f(\sigma_1)$. If S could have accepted <u>either</u> σ_1 <u>or</u> σ_2, then the subsequent behaviour of f(S) is defined <u>either</u> by $S(\sigma_1)$ <u>or</u> by $S(\sigma_2)$ - but the choice between them is nondeterministic. This nondeterminism is introduced by taking the <u>union</u> of the sets:

$$
\begin{aligned}
f(S) &=_{df} \{\surd\} && \text{if } S = \{\surd\} \\
&=_{df} \text{can } (\sigma : \{f(X) \mid X \in S^o\} \rightarrow \bigcup \{S(\delta) \mid \sigma = f(\delta)\}) && \text{otherwise}
\end{aligned}
$$

where $f(X) = \{f(\sigma) \mid \sigma \in X\}$

2.3. Concealment of internal communication.

Consider a parallel command S, containing processes with names in a set Λ:

$$\Lambda = \{a_1, a_2, \ldots, a_n\}$$

$$S = (a_1 . S_1 \| a_2 . S_2 \| \ldots \| a_n . S_n)$$

According to the construction given in [ ], all the symbols communicated within this command are prefixed by the name of the recipient and by the name of the sender, i.e., they all have the form "$a_i.a_j.\sigma$." Let Π be the set of such symbols:

$$\Pi = \text{loc}(\Lambda) =_{df} \{x.y.\sigma \mid x,y \in \Lambda \;\&\; x \neq y\}$$

We wish to define the result $S\backslash\Pi$ of concealing all the internal communications within S. Clearly, its alphabet excludes Π:

$$\Sigma(S\backslash\Pi) = \Sigma(S) - \Pi$$

The clearest way of defining $S\backslash\Pi$ is by giving the set of environments in which it may successfully be used. All internal communications are concealed from the environment simply by removing the relevant symbols from its alphabet.

$$S\backslash\Pi =_{df} \text{imp } \{Q \mid \Sigma(Q) = \Sigma(S) - \Pi \;\&\; Q \text{ is finite } \& \;\forall P.\; P \in Q \rightarrow \text{safe } (P||Q)\}$$

The notation $S\backslash\Pi$, but not its definition, is due to Robin Milner.

2.4. Localisation of process names.

After concealment of local communications between named subprocesses of a parallel command S, it ~~still~~ remains to deal with their nonlocal communications, which are still prefixed by the name of the participating process. We wish to strip off these local names, so that S itself, rather than its subprocesses, appears to be the participant in the communication. This stripping is achieved by a function

$$\text{strip}_\Lambda(x.\delta) = \delta \qquad \text{if } x \in \Lambda$$

$$\text{strip}_\Lambda(\sigma) = \sigma \qquad \text{if } \sigma \text{ is } \underline{\text{not}} \text{ prefixed by an } x \text{ in } \Lambda$$

Thus we define the parallel command of [ ]

$$[a_1::S_1||a_2::S_2||\ldots||a_n::S_n] =_{df}$$
$$\text{strip}_\Lambda((a_1.S_1||a_2.S_2||\ldots||a_n.S_n)\backslash\Pi)$$

where $\Lambda = \{a_1, a_2, \ldots, a_n\}$

and $\Pi = \{x.y.\sigma \mid x, y \in \Lambda \;\&\; x \neq y\}$

3. New operators

This section introduces new operators for non-deterministic processes, and illustrates their use in the design of modules for a simple operating system.

3.1. Alternative composition.

A non-deterministic version of alternative composition is based on Dijkstra's if construct [ ]. If S and T are processes (not containing ✓) then $S \square T$ on its first step accepts any symbol acceptable to either of them. If the symbol actually offered is acceptable by both of them, it is nondeterministic which one of them has participated in the communication. Otherwise, the one that can accept it does so.

$$S \square T =_{df} \mathrm{can}\ (\sigma:\{X \cup Y \mid X \in S^{o} \& Y \in T^{o}\} \rightarrow S(\sigma) \cup T(\sigma))$$

(Recall that $S(\sigma) = \{\}$ if $\sigma \notin \bigcup S^{o}$)

This operator is associative and commutative, and

$$(S \square T);R = (S;R) \square (T;R)$$

It is also idempotent on canonical forms.

Examples:

$\square$ can be used with the same effect as $\cup$, wherever $\cup$ is defined (i.e. when $(\bigcup S^{o}) \cap (\bigcup T^{o}) = \{\}$).

3.2. A server/master relationship.

It is convenient to introduce an asymmetric relation between a server process S and a master process T. The server process is given a name "s", by which T can communicate with it; but T has no name: all communications of S are directed towards T, except those which are outside the alphabet of T. The required definition is:

$$[s::S||T] =_{df} \mathrm{strip}_{\{s\}}((s.S||T)\setminus\pi)$$

where $\pi = \Sigma(T) \cap \Sigma(s.S)$

Example: arithmetic expression.

An arithmetic expression can be modelled as a process which inputs the values of its constituent variables (which are also processes), and outputs the value of the expression (e.g. a natural number) to its master. A constant is a process that merely outputs its value and stops, e.g.

$$(3 \rightarrow \checkmark)$$

A variable "p" inputs a value from p, and outputs it again:

(p.fetch?n:NN → n)

If S and T model subexpressions e and f, their sum "e+f" can be modelled by:

[left::S || [right::T || PLUS]]

where the master process PLUS is

(left?n:NN → (right?m:NN → m+n))

Note that Σ(PLUS) = NN ∪ left.NN ∪ right.NN,

so that PLUS does <u>not</u> participate in the communication between the server processes and their operands.

An assignment "p: = e" can be modelled by

[rhs::S || (rhs?n:NN → p.assign.(n))]

where S models e.

This example is due to John Kennaway.

3.3. Sequential iteration.

We define a sequential iteration S <u>until</u> T as a process which involves none or more executions of S followed by T :

S;S;...;S;T .

More precisely, S <u>until</u> T accepts any symbol acceptable by either S or by T; but it is actually accepted by only one of them. If it is accepted by S,S will proceed to completion, and then the whole construct will be repeated. If it is accepted by T, T will proceed to completion, and then the iteration will also terminate. The construct is defined formally by recursion:

[S <u>until</u> T] $=_{df}$ (S;[S <u>until</u> T]) $\Box$ T

A version of the conventional <u>for</u> statement can also be defined:

$$\sum_{i=m}^{n} S_i =_{df} \{\surd\} \qquad \text{if } m>n$$

$$=_{df} S_m; S_{m+1}; \ldots; S_n \qquad \text{otherwise.}$$

subscript n

Examples. (1) A subroutine OUT

A subroutine OUT accepts lines of 20 characters, and outputs them one at a time to a hardware process CONSOLE. Each line is followed by a "$\mathcal{L}$" symbol. OUT terminates on receipt of an "end" signal:

$$OUT =_{df} [s::CONSOLE \;||\; \underline{*}[\ell:LINE \rightarrow (\sum_{i=1}^{20} s.(\ell_i));\ s.\mathcal{L}\ \underline{until}\ end]\ ]$$

where LINE = $char^{20}$
and ℓ_i is the ith character of line ℓ.

This subroutine runs as a server in parallel with its master:

[t::OUT || (MASTER;t.end)]

The master sends a line ℓ to t by "t.ℓ". It cannot directly communicate with the hardware CONSOLE, since the name "s" of the console is local to OUT.

(2) A function F.

A function can be modelled by a process which inputs its parameters at the beginning, and outputs its results at the end:

F = (p:NN → ... compute and output result...)

If this function is to be invoked many times, it needs to be repeated; and if it is to be called by a master process T, it needs a name f:

[f:: *[F until end] || (T;f.end)]

A function call from within T is a special case of an arithmetic expression, and can be modelled by a process as described in 3.2. Let S be the process which models computation of the actual parameter of the call. Then the complete call is modelled by:

[param::S

||(param?n:NN → f.n;(f?r:NN → r))

]

(3) A Boolean semaphore SEM.

A Boolean semaphore has alphabet {P,V,end}. P represents acquisition of a single resource, and V represents its release. These symbols must alternate until the "end" is reached:

SEM = *[(P → V) until end]

(4) A batch processing system BPS.

A batch processing system accepts and executes a sequence of jobs presented on cards. Each job is preceded by a JOBCARD, and the last card indicates "end of batch".

BPS = *[j:JOBCARD → ... process next job ...
until end of batch]

3.4. Disjoint processes.

When two processes running in parallel are intended to communicate with their common environment but not with each other, they are said to be disjoint. If their alphabets are disjoint, the required effect can be achieved by the normal parallel operator ||. But if their alphabets are the same, or overlap, a more elaborate construct is required. The case is sufficiently common to deserve a special notation:

$$S|||T =_{df} [a::S||b::T] \qquad \text{(see 2.4.)}$$

where a and b are arbitrary distinct local process names. This operator is clearly associative and commutative.

It is also convenient to introduce a parallel analogue of the sequential "for statement"

$$\mathop{\Pi\!\Pi\!\Pi}_{i=m}^{n} S_i = \{\checkmark\} \qquad \text{if } n<m$$

$$= S_m|||S_{m+1}|||\ \dots\ |||\ S_n \quad \text{otherwise.}$$

A parallel form of iteration may also be defined:

[S until T] $=_{df}$ (S|||[S until T]) □ T

This ~~consists of~~ involves none or more disjointly parallel activations of S together with one activation of T:

S|||S||| ... |||T

Each new activation of S is triggered by acceptance of a symbol on the first step of S. The new activation of S runs in parallel with all previously triggered activations. After acceptance of an initial symbol by T, no further activations of S are possible. The parallel iteration terminates after termination of T and of all the previous activations of S.

Examples: (1) a more general semaphore.

A general semaphore initialised to value n represents a set of n identical resources, any one of which may be acquired by P and released by V. The P's and V's are interleaved, subject only to the constraint that the number of P's must never exceed the number of V's by more than n.

$$GSEM(n) = \prod_{i=1}^{n} SEM$$

This example is due to Robin Milner.

(2) A multiprogrammed batch system.

A multiprogrammed batch system contains a number of subprocesses, each of which is capable of running a series of jobs. These sub-processes do not communicate with each other, but they do communicate with other processes of the operating system in which they are embedded. For example, they may all send lines to the process t.OUT, (3.3. (1)). Such communications are arbitrarily interleaved.

$$MPBS(n) = \prod_{i=1}^{n} BPS.$$

(3) Two shared consoles.

In order to reduce the bottleneck of sharing a single console in a multiprogramming system, a second CONSOLE is introduced, exactly like the first. When a line is sent for output, it does not matter which console is used - whichever is free first will be satisfactory. The required effect is achieved by running two copies of OUT in parallel:

[t::(OUT|||OUT) || MPBS(n); t.end; t.end]

Note that no change is needed to OUT or to MPBS to accommodate introduction of the second console. The second "t.end" is required to terminate the new instance of OUT.

3.5. Channels.

The definition of parallel iteration given in the previous section provides a method of triggering a number of activations of a process S, which run in parallel with each other. Unfortunately, this definition is not very useful, because it is impossible to communicate with some particular activation of S; any symbol communicated with the parallel iteration may be accepted by <u>any</u> of the activations already triggered. What we require is that each new activation of S comes equipped with a new <u>channel</u>, one end of which is connected to the activation, and the other to the process which triggered that activation. The channel is then used by each process to achieve communication with the other. Each process should be able to declare its own (different) local name for the channel.

To achieve the required effect, we use a denumerably infinite set CHAN of channels (the natural numbers will serve this purpose if required). In order to "acquire" a fresh channel, a process S should simply "input" it:

$$(x\text{:CHAN} \rightarrow \ldots\ x.5\ \ldots\ x?n\text{:NN}\ldots)$$

The local name "x" stands for the channel, and is used for communication rather like a process name. Suppose another process T is running in parallel with S, and contains an exactly similar command:

$$(\underline{y}\text{:CHAN} \rightarrow \ldots \underline{y}?n\text{:NN}\ \ldots\ y.3\ \ldots)$$

If both S and T are ready to execute these commands simultaneously, the effect will be that <u>both</u> will "input" the <u>same</u> arbitrary member of CHAN. In S this will be bound to x and in T it will be bound to $\underline{y}$. Nevertheless, because x and $\underline{y}$ denote the same channel, every communication on x by S will match a communication on $\underline{y}$ by T.

In general, more than one channel will be required between S and T, and we must ensure that any newly acquired channel is distinct from all previously acquired ones. This can be achieved by a third process GEN, whose sole task is to allocate fresh channel names.

$$\text{GEN} =_{df} \text{GEN(CHAN)}$$

$$\text{where GEN}(X) = ((x\text{:}X \rightarrow \text{GEN}(X-\{x\}))$$
$$\square\ \text{endgen})\quad \text{for } X \subset \text{CHAN}$$

GEN runs in parallel with both S and T, and participates together with both S and T in their acquisition of new channels. Thus we define a more powerful version of the server/master relation:

$$[s::S \# T] =_{df} [s::(((S;\text{endgen})\,||\,\text{GEN})\backslash\{\text{endgen}\})\,||\,T]$$

Now, when T (or a subprocess of T) has to establish a new channel with a subprocess of S, it is necessary to quote the name "s":

(y:s.CHAN → ...y?n:NN... y.3...)

The formal definition of # is unpleasantly complicated. However, the dynamic establishment and disestablishment of distinct channels is no more unfamiliar than making a telephone call; it indicates that many separate calls can proceed in parallel; and it permits a clear expression of important programming techniques.

Example (1) a reentrant function.

The function F described in 3.3 (2) can be shared among many subprocesses of its master process T. However, since it is implemented by sequential iteration, only one subprocess can use it at a time. If the computation of the function is timeconsuming, this could be a bottle-neck in a multiprocessing system. The bottle-neck can be relieved if parallel iteration is used instead of sequential, since each activation can then proceed in parallel with those that are still running. But we need to ensure that the result of each activation is sent back to the same subprocess of T which sent the parameter for that activation. This is achieved by setting up a fresh channel, and using it exactly twice - once for the parameter and once for the result. Thus the body of a reentrant function looks like:

F = (ω:CHAN → (ω?p:NN → ...;ω.r))

This is combined with T :

[f:: <u>**</u>[F <u>until</u> end]#(T;f.end)]

A call of f from T with parameter E is modelled:

[param::E

||(y:f.CHAN → (param?n:NN → y.n);   (f!E?r → S(r))

(y?r:NN → r))

]

(2) Virtual line printers.

A process HLP describes the behaviour of a simple line printer, implemented in hardware. Its alphabet consists of lines of 125 characters each (i.e. the type LINE), and a symbol "throw", which positions the paper at the bottom of the next even-numbered page. The line printer is to be shared among the processes of a multiprogrammed batch processing system. Each process uses it on occasion for output of a file consisting of many lines. The lines from files output by separate processes must not be interleaved. To assist in separation of files, each one must begin and end on the turn of an even-numbered page with a line of asterisks.

A process acquires and uses the lineprinter thus:

```
(report: vℓp.CHAN → ...;report.line1;...; report.end)
```

where report is the local name for the file.
vℓp is the name of a process which implements virtual line printers.
line1 is an array of characters to be output
end indicates the end (closure) of the file.

The overall structure of the solution is

```
[vlp::  [hlp::HLP||VLP] ⋕ (T; vlp.end)]
```

```
where VLP  = hlp. throw; hlp.asterisks;
           *[user:CHAN → hlp.asterisks;
                 *[user?ℓ:LINE → hlp.ℓ
                  until user.end];
                  hℓp.throw; hℓp.asterisks
             until end];
```

(3) Two line printers.

To accommodate an increased load of printing a second HLP is installed. It is to be brought into use without making any change to T or to VLP. The solution is left as an exercise (hint. see 3.4 (3)).

(4) QUEUE

A queue is a process with alphabet

$$\Sigma(\text{QUEUE}) = \text{CHAN} \cup \text{CHAN.leave} \cup \{\text{serve}, \text{end}\}$$

A process waits on a queue named "q" by

(q?c:CHAN → c.leave)   c: q.CHAN

A server ends the wait of longest waiting process by

q.serve
~~which is itself delayed if the queue is empty.~~
The "end" signal may be given only when the queue is empty.

$QUEUE = Q_{\varepsilon}$

where $Q_{\varepsilon} = [u{:}CHAN \rightarrow Q_{<u>} \,\square\, end]$

$Q_{S} = [u{:}CHAN \rightarrow Q_{S<u>}$ if $S \neq \varepsilon$

$\square\, serve \rightarrow (S_{o}).leave;\ Q_{S'}]$

where S is a sequence of channels.

ε is the empty sequence

<u> is the sequence containing only u

S_{o} is the first item of S

S' is the rest of S, on removal of S_{o}

$S_{<u>}$ ~~is S~~ with u appended.

(raise)