A MODEL FOR

COMMUNICATING SEQUENTIAL PROCESSES

C.A.R. HOARE

Technical Monograph PRG-22
June 1981

Oxford University Computing Laboratory
Programming Research Group
45 Banbury Road
Oxford OX2 6PE

ABSTRACT

A previous paper (Hoare, 1978b) has suggested that parallel composition and communication should be accepted as primitive concepts in programming. This paper supports the suggestion by giving a simplified mathematical model for processes, using traces (Hoare, 1978a) of the possible interactions between a process and its environment.

<u>CONTENTS</u>

A MODEL FOR COMMUNICATING SEQUENTIAL PROCESSES

INTRODUCTION

The primary objective of this paper is to give a simple mathematical model for communicating sequential processes. The model is illustrated in a wide range of familiar programming exercises, including an operating system and a simulation study. As the exposition unfolds, the examples begin to look like programs, and the notations begin to look like a programming language. Thus the design of a language seems to emerge naturally from its formal definition, in an intellectually pleasing fashion.

The model is not intended to deal with certain problems of non-determinism. These have been avoided by observance of certain restrictions detailed in the technical notes. No attention has been paid to problems of efficient implementation; for this, even further restrictions should be imposed.

The long term objective of this study is to provide a basis for the proof of correctness of programs expressed as communicating sequential processes. However, in this paper the formalities have been kept to a minimum and no proofs are given.

Concepts

(1) The ultimate constituent of our model is a symbol, which may be intuitively understood as denoting a class of event in which a process can participate.

(a) "5p" denotes insertion of a coin into the slot of a vending machine VM.

(b) "large" denotes withdrawal from VM of a large packet of biscuits.

(c) "up" denotes incrementation of a COUNT register.

(2) The alphabet of a process is the set of all symbols denoting events in which that process can participate.

(d) {5p, 10p, large, small, 5pchange} is the alphabet of the vending machine VM.

(e) {up, down, iszero} is the alphabet of COUNT.

(3) A trace is a finite sequence of symbols recording the actual or potential behaviour of a process from its beginning up to some moment in time.

(f) <10p, small, 5pchange> is a trace of a successful initial transaction of VM.

(g) < > (the empty sequence) is a trace of its behaviour before its first use.

(h) <up, down, iszero, down> is not a trace of a COUNT, since a zero count cannot be decremented.

(4) A process P is defined by the set of all traces of its possible behaviour. From the definition of a trace, it follows that for any process P,

(1) < > is in P (i.e., P is non-empty)

(2) if st (the concatenation of s with t) is in P then so is s by itself (i.e., P is prefix-closed)

These properties will help to simplify the definition of parallel composition of processes.

Notations

(1) The process ABORT is one that never does anything.

$\text{ABORT} = \{\langle\ \rangle\}$

(2) If c is a symbol and P is a process the process $(c \rightarrow P)$ first does "c" and then behaves like the process P.

$$(c \rightarrow P) = \{\langle\ \rangle\} \cup \{\langle c \rangle s \mid s \text{ is in } P\}$$

where $\langle c \rangle$ is the sequence consisting solely of the symbol c. By convention the arrow associates on the right, so that

$$c \rightarrow d \rightarrow P = c \rightarrow (d \rightarrow P).$$

(3) The process $P \square Q$ behaves either like the process P or like the process Q; the choice will be determined by the environment in which it is placed.

$$P \square Q = P \cup Q \qquad \text{(normal set union)}$$

(See technical note (1).)
By convention $\rightarrow$ binds more tightly than $\square$, so that

$$c \rightarrow P \square d \rightarrow Q = (c \rightarrow P) \square (d \rightarrow Q).$$

(4) The alphabet of a process P will be denoted by $\overline{P}$. Usually we will assume that the alphabet of a process is given by the set of all symbols occurring in its traces.

$$\overline{\text{ABORT}} = \{\} \qquad \text{(the empty set)}$$

$$\overline{c \rightarrow P} = \{c\} \cup \overline{P}$$

$$\overline{P \square Q} = \overline{P} \cup \overline{Q}$$

(5) We shall frequently use recursive definitions to specify the behaviour of long-lasting processes. These recursions are to be understood in the same sense as the recursive equations of (say) a context-free grammar expressed in BNF.

Examples

```
(i) VM = (5p→(5p→(large→VM□5p→ABORT)
              □small→VM
              )
          □10p→(small→(5pchange→VM)
              □large→VM
          )   )
```

On its first step VM accepts either 5p or 10p. In the first case, its following step is either the acceptance of a second 5p (preparatory to withdrawal of a large packet of biscuits) or the immediate withdrawal of a small packet. The second case should be self-explanatory. In all cases, after a successful transaction, the subsequent behaviour of VM is to offer a similar service to an arbitrary long sequence of later customers. But if any customer is so unwise as to put three consecutive 5p coins into the slot, the machine will break (ABORT), and never do anything else again.

In a conventional BNF grammar, the use of mutually recursive definitions is familiar. To avoid the limitations of context-free languages, we shall sometimes give an infinite set of mutually recursive definitions.

(j) $COUNT_n$ describes the behaviour of a count register with current value n. For n>0,

$$COUNT_n = (up \rightarrow COUNT_{n+1} \square down \rightarrow COUNT_{n-1})$$

whereas the behaviour of a zero count is

$$COUNT_0 = (up \rightarrow COUNT_1 \square iszero \rightarrow COUNT_0).$$

A zero count cannot be decremented, but it can respond to a test "iszero". The use of this test will be illustrated later.

PARALLEL COMBINATION OF PROCESSES

The traces of a process define all its possible behaviours. The actual behaviour of a process P operating in an environment E will in general be constrained by this environment. The environment E can also be defined as a process, consisting of all sequences of events in which it is capable of participating. Each event that actually occurs must be possible at the time of occurrence for both the process and for its environment. Consequently, the set of all the traces of the process and its environment operating in parallel with each other is simply the intersection of the two sets (P∩E).

For example, a customer of a vending machine is initially prepared to accept a large or even a small packet of biscuits, if they are available. Alternatively he inserts a coin, without noticing its value, and then attempts to withdraw a large packet of biscuits.

CUSTOMER = {< >,<large>, <small> ,<10p>, <5p>,
<10p,large> , <5p,large>}

When VM interacts with this customer, the set of possible traces of their interaction is

VM||CUSTOMER = {< >,<10p>,<10p,large> <5p>}

Note how VM does not permit the customer to withdraw the biscuits without paying. But even worse, after insertion of 5p the VM is prepared to yield only a small packet of biscuits, whereas the foolish customer is trying vainly to extract a large packet. No further events are possible; machine and customer are locked forever in deadly embrace (Dijkstra, 1968).

The description given above assumes that the alphabets of the process and its environment are the same, so that every event requires simultaneous participation of both of them. In general, some of the symbols could be in the alphabet of only one of the two processes, and so the corresponding events can occur without the participation of the other process. For example, a customer may fumble in his pocket, or curse when he is thwarted; a vending machine may clink on accepting a coin and clunk on withdrawal of biscuits.

CUSTOMERB = {<fumble,5p,large>, ...
<fumble,5p,curse,small>, ...}

NOISYVM = {<5p, clink, small, clunk>...}

Events which are particular to only one of the interacting processes can occur concurrently with events particular to the other one. It is convenient to model such concurrency by arbitrary interleaving of symbols. Thus the traces of the combined behaviour of NOISYVM and CUSTOMERB will include

{<fumble,5p,clink,curse,small,clunk>,
<fumble,5p,curse,clink,small,clunk>, ...}

even though the clink and the curse can overlap in real time. The reason why interleaving is an acceptable model of concurrency is that we are interested only in the logical properties of processes and not in their timing.

The process (P||Q) is the process resulting from the operation of P and Q in parallel. The curious mixture of synchronisation of symbols in both their alphabets with interleaving of the other symbols has a surprisingly simple definition.

$$(P||Q) = \{s \mid s \text{ is in } (\overline{P} \cup \overline{Q})^{*} \ \& \ (s \upharpoonright \overline{P}) \text{ is in } P \ \& \ (s \upharpoonright \overline{Q}) \text{ is in } Q\}$$

where $s \upharpoonright X$ (s restricted to X) is obtained from s by simply omitting all symbols outside X,

and X^{*} is the set of finite sequences of symbols from X.

Thus each process ignores events of the other process which do not require its participation. In the case that the alphabets of the two processes are the same, (P||Q) is just the intersection of the sets (P∩Q). In the case where the alphabets are disjoint ($\overline{P} \cap \overline{Q} = \{\}$), (P||Q) is the set of all interleavings of a trace from P with a trace from Q. We adopt the convention that || is the most loosely binding operator.

A well-known example on which to test this definition is the story of the five dining philosophers. The system as a whole consists of two groups of processes:

DINING ROOM = PHILOSOPHERS||FORKS

where PHILOSOPHERS = $PHIL_0$||...||$PHIL_4$

and FORKS = $FORK_0$||...||$FORK_4$

and $PHIL_i$ = (i sitsdown →
i picksup fork i →
i picksup fork (i ⊕ 1) →
i putsdown fork i →
i putsdown fork (i ⊕ 1) →
i getsup →
$PHIL_i$)

and $FORK_i$ = (i picksup fork i → i putsdown fork i → $FORK_i$
⊓(i ⊖ 1) picksup fork i →
(i ⊖ 1) putsdown fork i → $FORK_i$)

where (i ⊕ 1), (i ⊖ 1) are taken modulo 5.

The alphabets of the philosophers are pairwise disjoint. This means that (characteristically) they do not interact directly with each other: their joint behaviour is an arbitrary merging of their individual behaviours. The same is true of the forks. However, each event of picking up a fork and putting it down requires simultaneous participation of exactly two processes, one philosopher and one fork.

It is well known that the simple system described above is liable to a deadly embrace after:

<0 sitsdown,..., 4 sitsdown,
0 picksup fork 0,..., 4 picksup fork 4>.

An ingenious solution to this problem is to introduce a BUTLER process into the dining room; his task is to assist each philosopher to and from his seat, ensuring as he does so that not more than four philosophers are seated at a time.

NEWDININGROOM = DININGROOM||$BUTLER_0$

where $BUTLER_n$ (for n between 0 and 4) describes the behaviour of the butler when there are n philosophers seated. For example

$BUTLER_4 = (0\ getsup \rightarrow BUTLER_3 \sqcap \ldots \sqcap 4\ getsup \rightarrow BUTLER_3)$

The remaining cases will be defined later.

SEQUENTIAL COMBINATION OF PROCESSES

The process ABORT has been defined as one that never does anything, because it is already broken. We now wish to introduce another process SKIP, which also does nothing, but for a completely different reason: it has already succeeded, and there is nothing more for it to do. Successful termination can be regarded as an event denoted by a special symbol ✓ (success), and the process that just succeeds is:

SKIP = {<>,<✓>}.

(See technical note (2).)

The use of SKIP can be illustrated by adapting some previous examples.

(a) A vending machine which participates in just one transaction (successful or unsuccessful):

```
VM1 = (5p → (5p → (large → SKIP ▯ 5p → ABORT)
                 ▯ small → SKIP)
         ▯ 10p → (small → (5pchange → SKIP)
                 ▯ large → SKIP))
```

(b) A customer, who terminates successfully after a single successful transaction:

```
CUSTOMERC = (5p → large → SKIP
             ▯ 10p → large → SKIP)
```

(c) Their joint behaviour is:

```
VM1||CUSTOMERC = (5p → ABORT
                 ▯ 10p → large → SKIP)
```

Note that when ✓ is in the alphabet of both P and Q, successful termination of (P||Q) requires that both of them terminate successfully. (See technical note (3)).

The introduction of the concept of successful termination permits the definition of sequential composition (P;Q) of processes P and Q.

This behaves first like P. If P fails, then so does (P;Q). But if P has terminated successfully, (P;Q) continues by behaving like Q. More formally,

$$P;Q = \{s \mid s \text{ is in } P \text{ and } s \text{ does not contain } \checkmark\}$$
$$\cup\{st \mid s\langle\checkmark\rangle \text{ is in } P \text{ and } t \text{ is in } Q\}$$

We adopt the convention that semicolon binds most tightly, so that

$$a \rightarrow P;Q \mathbin{\Box} R = a \rightarrow (P;Q) \mathbin{\Box} R$$

A simple repetitive statement can be defined

$$\underline{\text{for}}\ i{:}\ell..h \rightarrow P_i = \text{SKIP} \qquad \text{if } h<\ell$$
$$= P_\ell ; P_{\ell+1} ; \ldots ; P_h \qquad \text{if } \ell \le h$$
$$P\ \underline{\text{until}}\ Q = Q \mathbin{\Box} (P;\ (P\ \underline{\text{until}}\ Q))$$

(d) A vending machine which serves at most three customers:

$$\text{VM3} = \text{VM1};\ \text{VM1};\ \text{VM1}$$

(e) And now twenty customers:

$$\text{VM20} = \underline{\text{for}}\ i{:}1..20 \rightarrow \text{VM1}$$

(f) An automaton which accepts any number of "a"s followed by a single "b" and then the same number of "c"s:

$$A^nBC^n = (b \rightarrow \text{SKIP} \mathbin{\Box} (a \rightarrow (A^nBC^n;\ (c \rightarrow \text{SKIP}))))$$

(g) A process which accepts any interleaving of more "up"s than "down"s; but terminates successfully on first receiving one more "down" than "up":

$$\text{POS} = (\text{down} \rightarrow \text{SKIP} \mathbin{\Box} \text{up} \rightarrow (\text{POS};\text{POS}))$$

Note: to counteract an initial "up" it is necessary to accept <u>two</u> more "down"s than "up"s; this is done by first accepting one more, and then by accepting one more again.

(h) An alternative formulation of (g):

$$\text{POS} = (\text{up} \rightarrow \text{POS})\ \underline{\text{until}}\ (\text{down} \rightarrow \text{SKIP})$$

(i) A process that behaves exactly like COUNT_0:

$$\text{ZERO} = (\text{iszero} \rightarrow \text{ZERO} \mathbin{\Box} \text{up} \rightarrow (\text{POS};\ \text{ZERO}))$$

(j) An automaton that accepts equal numbers of "a"s, "c"s, and "e"s:

$A^nBC^nDE^n = (A^nBC^n; (d \rightarrow SKIP)) \| C^nDE^n$

where C^nDE^n will be defined below.

The first process ensures that the "c"s match the "a"s, and ignores the "e"s. The other process ignores the "a"s, but ensures that the "c"s are matched by the "e"s.

In future we shall often abbreviate

"$(d \rightarrow SKIP)$" to just "d"

ALPHABET TRANSFORMATION

Let f be a total function which maps the symbols of one alphabet Y onto symbols of another alphabet Z, so that:

f(y) is in Z for all y in Y

Given a process P with alphabet Y, we can define a process f(P) with alphabet Z, which behaves like P, except that it does f(y) whenever P would have done y.

$f(P) = \{f(s) \mid s \text{ is in } P\}$ (See technical note (4))

where f(s) is obtained from s by applying f to each of its symbols.

(a) to represent the sad effect of monetary inflation on a vending machine:

NEWVM = f(VM)

where f(5p) = 10p, f(small) = verysmall, etc.

(b) a process used in an earlier example

$C^nDE^n = f(A^nBC^n)$

where f(a) = c, f(b) = d, and f(c) = e

The most frequent use of alphabet change will be to give different <u>names</u> to otherwise similar processes. So we introduce a set M of special symbols to serve as process names. If x denotes an event, and m is a name in M, then the compound symbol "m.x" denotes participation in event x by a process named m. We stipulate that events prefixed by distinct process names are distinct:

$m \neq n$ implies $m.x \neq n.x$

with the exception that $m.\checkmark = \checkmark$ for all names m.

The prefixing of a name is accomplished by a function

$$\text{prefix}_m(x) = m.x \qquad \text{for all } x.$$

We can now define m:P as a process with name m, which does m.x whenever P would do x:

$m{:}P = \text{prefix}_m(P)$ (By convention $m{:}\ P;Q||R = (m : (P;Q))||(R)$)

(c) Two distinct vending machines, operating independently in parallel (by interleaving of traces):

$(\text{red:VM} \mid\mid \text{green:VM})$

In general, the alphabet of a process will contain (in addition to events that require participation of its external environment) certain other events which represent its internal workings. These internal events are intended to occur automatically, without participation or even knowledge of the environment. To model the concealment of such events, we wish to remove the corresponding symbols from the alphabet of the process, and from every trace of its behaviour. Let X be the set of symbols to be concealed; the result of the concealment is defined:

$P\backslash X = \{s\upharpoonright(\overline{P}-X) \mid s \text{ is in } P\}$ (See technical note (5))

where $\overline{P\backslash X} = \overline{P}-X$ (set subtraction)

(d) A soundproofed version of NOISYVM

NOISYVM $\backslash$ **{clink, clunk}**

When a process has been defined by parallel composition of two or more processes, the mutual interactions of the component processes are often of no concern to their common environment. These interactions are just the events named by symbols occurring in the alphabets of more than one of the components. We represent the concealment of these events by enclosure in square brackets:

$$[P||Q] = (P||Q) \backslash (\overline{P}\cap\overline{Q}) - \{\checkmark\}$$

This definition generalises to more than two components:

$$[P_1||P_2||\ldots||P_n] = (P_1||P_2||\ldots||P_n) \backslash X - \{\checkmark\}$$

where $X = \bigcup_{i\neq j}(\overline{P}_i \cap \overline{P}_j)$

(e) A USER process uses a COUNT register named m, interacting with it by events

{m.iszero, m.up, m.down}

These interactions are to be concealed, thereby ensuring that the register serves as a local variable for the benefit of only the single user:

$[m:COUNT_0 \| USER]$

(f) Similar to (e), but with two registers:

$[n:COUNT_3 \| m:COUNT_0 \| USER]$

(g) Inside the USER process, the following subprocess will add the current value of n to m, leaving the value of n unchanged:

```
ADDNTOM = (n.iszero → SKIP
           □ n.down → ((m.up → ADDNTOM) ;
                       (n.up → SKIP)
                       )
           )
```

Another use for concealment is to remove $\surd$ from the alphabet of a process that is not intended to terminate. For example, if P is a normally terminating process, $\underline{*}P$ is a process which repeats P for as long as is required by the environment within which it runs:

$\underline{*}P = (P; (\underline{*}P)) \setminus \{\surd\}$

(h) A familiar example:

$VM = \underline{*}VM1$

INPUT AND OUTPUT

The model developed in the previous sections is sufficiently general to apply to any kind of event. In the following sections we shall be concerned primarily with communication events, involving output of information by one process and input of information by another. For these events we introduce particular notations. If t is a value of type T, then

!t denotes output of a message with value t

?t denotes input of a message with value t.

(a) A process which behaves as a Boolean variable. At any time, it is ready to input its next value or to output the value which it has most recently input (if any).

$$\mathrm{BOOL} = (?\mathrm{true} \rightarrow \mathrm{TRUEBOOL} \;\square\; ?\mathrm{false} \rightarrow \mathrm{FALSEBOOL})$$

$$\mathrm{TRUEBOOL} = (?\mathrm{true} \rightarrow \mathrm{TRUEBOOL} \;\square\; ?\mathrm{false} \rightarrow \mathrm{FALSEBOOL} \;\square\; !\mathrm{true} \rightarrow \mathrm{TRUEBOOL})$$

and FALSEBOOL is similar.

When a process performs input of some value x, its subsequent behaviour will usually depend on the value which it has just input. Although the type T of x may be known, the identity of the value which is actually going to be input is usually *not* known; the process must be prepared to do ?t (input of t) for *any* t in T; the selection will be made by its environment. To achieve this we introduce a form of input command:

$$(?x{:}T \rightarrow P_x) = \{\langle\rangle\} \cup \{\langle ?t\rangle s \mid \text{for } t \text{ in } T \text{ and } s \text{ in } P_t\}$$

Note that the variable x is a dummy (bound, local) variable in this construction. It is *not* a symbol, and it does *not* appear in any traces of the process.

(b) A process which just copies what it inputs:

$$\mathrm{COPY}_T = (?x{:}T \rightarrow !x \rightarrow \mathrm{COPY}_T)$$

This process serves as a one-place buffer.

(c) Similar to (b), except that consecutive pairs of "*" are replaced by "↑":

```
SQUASH = *(?x: CHAR →
              if x ≠ "*" then !x
              else (?y:CHAR → if y = "*" then ! "↑"
                       else ! "*" → !y))
```

(d) A process which behaves as a variable of type T:

$$\mathrm{VAR}_T = (?x{:}\,T \rightarrow \mathrm{VAR}_x)$$

where $\mathrm{VAR}_x = (!x \rightarrow \mathrm{VAR}_x \;\square\; (?y{:}T \rightarrow \mathrm{VAR}_y))$

VAR_x is the behaviour of a variable with value x.

Clearly, $\mathrm{BOOL} = \mathrm{VAR}_{\{\mathrm{false},\mathrm{true}\}}$

(e) A process which inputs cards, and outputs their contents one character at a time, interposing an extra space after each card:

$$\mathrm{UNPACK} = \underline{*}(?c{:}\mathrm{CARD} \rightarrow$$
$$(\underline{\mathrm{for}}\ i{:}\ 1..80 \rightarrow !c_i)\ ;\ !\text{" "})$$

where CARD = $\underline{\text{array}}$ 1..80 $\underline{\text{of}}$ CHAR.

(f) A process which inputs characters one at a time and assembles them into lines of 125 characters, which are then output

$$\mathrm{PACK} = \mathrm{PACK}_{<>}$$

where $\mathrm{PACK}_{\ell} = !\ell;\mathrm{PACK}_{<>}$ (if length $(\ell) = 125$)

$= (?c{:}\ \mathrm{CHAR} \rightarrow \mathrm{PACK}_{\ell<c>})$ (otherwise)

(g) A queue QUEUE_T at any time is prepared to input a new element of type T, or to output the element which was input the earliest (if any):

$$\mathrm{QUEUE}_T = \mathrm{BUFF}_{<>}$$

where $\mathrm{BUFF}_{<>} = (?x{:}T \rightarrow \mathrm{BUFF}_{<x>})$

and for $s \neq <>$,

$$\mathrm{BUFF}_s = (?x{:}T \rightarrow \mathrm{BUFF}_{s<x>}$$
$$\Box !\ \mathrm{first}(s) \rightarrow \mathrm{BUFF}_{\mathrm{rest}(s)})$$

(h) A stack is similar to a queue, except that it outputs the element which was input the latest; it can also give an indication when it is empty:

$$\mathrm{STACK}_T = \underline{*}(!\ \mathrm{isempty} \rightarrow \mathrm{SKIP}$$
$$\Box ?x{:}T \rightarrow \mathrm{STK}_x)$$

where $\mathrm{STK}_x = (?y{:}T \rightarrow \mathrm{STK}_y\ ;\ \mathrm{STK}_x\ \Box\ !x \rightarrow \mathrm{SKIP})$

is a stack with x as top value, which terminates when empty.

COMMUNICATION

Suppose that we wish two processes P and Q to operate in parallel in such a way that every message output by P is input directly by Q. The resulting compound process is denoted (P>>Q), which can be read: "P feeds Q". The synchronisation involved in direct communication requires that each output !t in P be regarded as the same event as an input ?t in Q. Such events are to be concealed from their common environment.

The required effect is achieved by transforming the alphabets of P and Q, prior to their composition. Thus we define

$$P{>>}Q = [\mathrm{strip}!(P) \| \mathrm{strip}?(Q)]$$

where strip!(!t) = t,   strip!(?t) = ?t

and   strip?(!t) = !t,   strip?(?t) = t

Note that all output from the outside environment is input by P, and all output by Q is input by the environment.

(a) Text is to be input from 80-column cards and output in lines of 125 characters each:

    LISTING = UNPACK>>PACK

(b) Similar to the above, except that consecutive "*"s are to be replaced by "↑":

    CONWAYS EXAMPLE = UNPACK>>SQUASH>>PACK

(c) Similar to (a) except that communication is desynchronised by interposing an unbounded buffer:

    UNPACK>>$\text{QUEUE}_{\text{CHAR}}$>>PACK

    This example shows that no generality is lost by taking synchronised communication as primitive.

(d) Similar to (c) but with only double buffering:

    UNPACK>>$\text{COPY}_{\text{CHAR}}$>>$\text{COPY}_{\text{CHAR}}$>>PACK

NAMED SOURCE AND DESTINATION

The >> combinator allows construction of chains of anonymous communicating processes, each taking input from its predecessor and sending output to its successor in the chain. For other more elaborate patterns of communication we shall use named processes, and allow each input or output to quote the name of its source or destination:

    m!t denotes output of message t to process named m

    m?t denotes input of message t from process named m.

(a) to update and test a Boolean variable named b: BOOL

    USERB = (...b!true ...(b?true → ... □b?false → ...)...)

    This has the effect(..b:=true...(if b <u>then</u>...<u>else</u>...)...)
    We also need to input arbitrary values from a named source:
    $(m?x{:}T \rightarrow P_x) = \{<>\} \cup \{<m?t>s \mid t \text{ is in } T \text{ and } s \text{ is in } P_t\}$

(b) to update an integer variable named **m**:

USERM = (... m!7 ... (m?x:INT → m!(x+3))...)

This has the effect: (... m:=7 ... m:=m+3 ...)

Henceforth we shall use these conventional notations for updating variables.

(c) a subroutine which repeatedly inputs a floating point argument and outputs its tangent as result:

TAN = $\underline{*}$(?x:FP → sin!x → cos!x →
        (sin?y:FP → (cos?z:FP → !(y/z))))

In order to establish synchronised communication between a named process m:P and an unnamed process Q, we need to ensure that each m!t in Q denotes the <u>same event</u> as ?t in P, and each m?t in Q denotes the <u>same event</u> as !t in Q. This is conveniently achieved by adapting the definition of prefix_m when applied to input and output events, thus:

$\text{prefix}_m(?t) = m!t$ and $\text{prefix}_m(!t) = m?t$.

In future we shall assume that this adapted definition of prefix_m is used in process naming.

As in the case of >>, it is usually desirable to conceal communications with a named process. We therefore define

$[m:P||Q] =_{df} (m:P||Q) \setminus \mathbf{m.X}$

where m.X is the set of all symbols prefixed by m.

(d) to declare a local Boolean variable for USERB:

[b:BOOL||USERB]

(e) to declare a local integer variable for USERM:

$[m:VAR_{INT}||USERM]$

(f) a subroutine which calls two local subroutines to assist in its calculations:

TANGENT = [sin:SIN||cos:COS||TAN]

(g) A subroutine which computes a factorial by recursion. As before, the argument and result are communicated by input and output:

```
FAC = (?x:NN → if x = 0 then !1
          else [f:FAC ||
                   f!(x-1);  (f?y:NN → !(x*y))])
```

Each activation, if necessary, creates another activation to compute the recursive call.

(h) A similar technique can be used to define a recursive data structure, for example, a set which inputs its members, and answers "!yes" if the value input was already a member and "!no" otherwise. Each activation stores one number x, and uses a recursive activation to store the rest of the set.

```
SET_T = (?x:T → !no →
                 [rest:SET_T ||
                *(?y:T → if y = x then !yes
                   else rest!y → (rest?yes → !yes
                                  □ rest?no → !no
                                  )
                )])
```

The previous examples show communication between a single named (slave) process and a single unnamed (master) process. In more general communication networks, it is necessary to allow one named process to communicate with another named process. As before, this is accomplished by equating the event m!t in a process named n with the event n?t in a process named m. Again, the definition of prefix_m is adapted for this purpose:

$$\text{prefix}_m(n?t) = \text{prefix}_n(m!t) = n.m!t.$$

(i) A network for multiplication of a matrix by a vector. Processes COL1, COL2, COL3 output the columns of a matrix IN. Values v_1, v_2, v_3 form a vector by which the matrix is to be multiplied. The resulting column is to be output to a DISPLAY process.

Since it is desirable to input three numbers at a time, and multiply three numbers at a time, a network of processes is required. They are pictured in figure 1, where each communication channel is annotated by the typical value that passes along it. Each of the processes M_1, M_2, M_3 repeatedly inputs a partial sum from above and a column value from the left, and then send its result down to its successor. The algorithm is defined:

$$[m_0:M_0||m_1:M_1||m_2:M_2||m_3:M_3||m_4:\text{DISPLAY}]$$

where $M_0 = \underline{*}(m_1!0)$ (a source of zeros) and for $0 < i \leq 3$

$$M_i = \underline{*}(m_{i-1}?sum:FP \rightarrow$$
$$col_i?x:FP \rightarrow m_{i+1}!\ (sum + v_i*x))$$

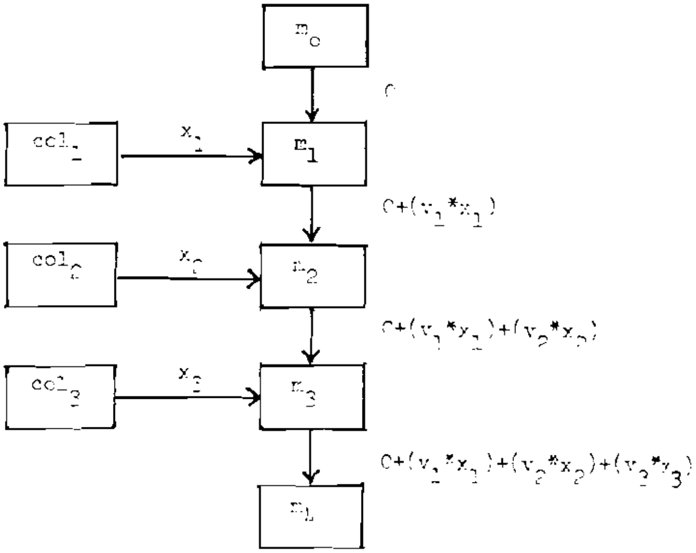

Figure 1

SHARING

Let X be a finite or infinite set, and let P_x be a process for each x in X.

$$[||x:X]P_x = \text{ABORT} \quad \text{if X is empty}$$
$$= P_u||P_v|| \ldots \quad \text{if X is } \{u,v, ..\}$$

(see technical note (6))

$$[\Box x:X]P_x = \text{ABORT} \quad \text{if X is empty}$$
$$= P_u \Box P_v \Box \ldots \quad \text{if X is } \{u,v, \ldots\}.$$

We define ANY as the set of <u>all</u> process names,
and any(r) = $\{r_i | i$ is an integer$\}$.

(a) PHILOSOPHERS = $[||i:0..4]\ PHIL_i$

(b) $BUTLER_n = [\Box i:0..4]\ (i\ sitsdown \rightarrow BUTLER_{n+1}$
$\Box i\ getsup \rightarrow BUTLER_{n-1})$ for n = 0..3

(c) An exclusion semaphore:

MUTEX = $\underline{*}$([□x:ANY]x?acquire → x?release)

It must be released by the same process which acquired it.

(d) An array of three exclusion semaphores, protecting three identical resources:

[||i:1..3]r_i::MUTEX

A user can acquire and release any one of the available resources by

([□mine:any(r)]mine!acquire → ...use the resource...;
mine!release)

(e) A hardware line printer with name h is to be shared for the output of complete files

LP_h = $\underline{*}$([□x:ANY]x?acquire →
(x?ℓ:LINE → h!ℓ <u>until</u> x?release → SKIP))

Each iteration of the major loop first "acquires" an arbitrary user x, and then copies lines from x to h, until receiving a "release" signal.

(f) This improved definition of LP_h ensures that each user's file is separated from the next by a "!throw" to the next even page boundary, and two rows of "!asterisks".

```
LP_h = (h!throw → h!asterisks →
      *([□x:ANY]x?acquire → h!asterisks;
         (x?ℓ:LINE → if ℓ≠asterisks then h!ℓ else SKIP
         until x?release);
         h!throw → h!asterisks)
```

(g) A shared variable of type T.

$SHAR_T$ = ([□x:ANY]x?y:T → SH_y)

where SH_y = ([□x:ANY]x!y → SH_y
□[□x:ANY]x?z:T → SH_z)

This example shows that a communication-based theory of parallelism is not in principle different from one based on shared variables.

In the previous examples, when many processes attempt simultaneously to acquire a shared resource, all but one will have to wait; and when the resource is released, it is not determined in what sequence they will eventually acquire the resource. If it is important to control the sequence of acquisition, we need a more complicated scheduler which will separate the request and the granting of the resource as distinct events.

(h) A "first-come first-served" scheduler, sharing a group of N resources. A QUEUE is needed to store the names of waiting users, and an integer to indicate the number of free resources minus the number of ungranted requests:

```
FCFS_N = [q:QUEUE_ANY||free:VAR_INT||
            free :=N;
           *([□x:ANY]x?request → free := free-1;
                if free < 0 then q!x else x!granted
           □[□x:ANY]x?release → free := free+1;
                if free ≤ 0 then (q?y:ANY → y! granted) else SKIP
           )]
```

Conventional notations have been used for updating variables.

A MULTIPROGRAMMED BATCH PROCESSING SYSTEM

A multiprogrammed batch processing system inputs jobs from any of C cardreaders, executes them on any of P processors, and outputs the results on any of L line printers. An account is kept of the cost of each job, and this is printed out at the end. If the cost exceeds a certain limit, the job is truncated.

The overall structure of the system is

MBPS = [CARDREADERS||LINEPRINTERS||PROCESSORS]

where CARDREADERS = [||i:1..C]cr_i:CR_{c_i}

and LINEPRINTERS = [||i:1..L]lp_i:LP_{h_i}

and PROCESSORS = [||i:1..P]pr_i:PROC.

Each processor executes a stream of jobs submitted by users:

PROC = *SINGLEJOB

The process SINGLEJOB executes a single user's job; taking input from any free reader and channelling output to any free printer :

```
SINGLEJOB =
[cost:$VAR_{NN}$||c:$VAR_{CARD}$||
([□in:any(cr)]in!acquire →
[□out:any(lp)]out!acquire →
      cost:=0; RUN;
      in!release;
      out!account(cost); out!release
)]
```

The process RUN needs an auxiliary process USER (not shown here) which actually executes the user's job. This USER is assured to be initialised to some standard compiler or control language interpreter. It interposes a regular "!timeslice" signal after every million instructions executed; and sends a "!finished" signal when the user program is finished (if ever):

```
RUN = [pr:USER||LOOP]
where LOOP = cost := cost + 1;
             if cost > costlimit then SKIP
             else (pr?ℓ:LINE → out!ℓ → LOOP
                  □pr!c → (in?x:CARD → c!x → LOOP)
                  □pr?timeslice → LOOP
                  □pr?finished → SKIP
                  )
```

In practice, the interface between USER and LOOP will be implemented by hardware protection mechanisms and by supervisor calls and exits.

In order to prevent interference between successive jobs submitted in a batch, the cards of each job are separated from the next job by an "endcard", which is used for no other purpose. The task of CR is to ensure that the cards for each job are consumed right up to the endcard but not beyond it:

```
$CR_h$ = *([□x:ANY]x?acquire → (h?c:CARD → $FILE_c$))
where $FILE_c$ = (x!c → if c = endcard then $FILE_c$
                                  else(h?c:CARD → $FILE_c$)
           □x?release → $SCAN_c$)
where $SCAN_c$ = if c = endcard then SKIP
                 else(h?c:CARD → $SCAN_c$)
```

If the user attempts to read beyond the endcard, he just gets further copies of the endcard.

We now specify an array of processes which perform pseudo-offline output of files. Each process uses a file (acquired from a filing system) to hold the user's output, and acquires a real line printer only when the user's output is complete.

```
SPOOLDLPS = [||i:NN]slp$_i$ : SLP
where SIP = [□x:ANY]x?acquire →
                [□f:any(file)]f!acquire →
                 (x?ℓ:LINE → f!ℓ until x?release);
                 f!rewind;
                 ([□out:any(lp)]out!acquire →
                    (f?ℓ:LINE → out!ℓ until f?eof);
                    out!release)
```

SLP acts like a "process" in a language like MODULA; a new "instance" comes into existence as a result of each "call" of the form:

```
([□out:any(slp)]out!acquire → ... out!ℓ1...out!ℓ2...out!release)
```

DISCRETE EVENT SIMULATION

In designing a program to simulate a fragment of the real world, it is necessary also to simulate the passage of real time. Any process of the program may need to enquire the current value of simulated time, by inputting it from a "timer" process:

```
(timer?t:TIME → ...t is time now...).
```

Furthermore, a process may need to delay itself until simulated time reaches some predetermined value, say 8 o'clock. This is done by outputting the required "alarm setting" to the timer process:

```
timer!8
```

This is an event which is guaranteed to occur only at 8 o'clock (in simulated time). Thus, to delay itself for d units of simulated time, a process can perform the actions:

```
HOLD(d) = (timer?t:TIME → timer!(t+d))
```

The timer process is always prepared to output the current value of simulated time. It is also prepared to input a value, provided that this is equal to the current value of simulated time; in this way a process performing a HOLD operation is permitted to continue. Finally, if all activity of the user processes has terminated, the simulated time clock is stepped on to its next value. TIM_t describes the behaviour of the timer at simulated time t:

$$\begin{aligned} TIM_t = (&[\Box x{:}ANY]x!t \rightarrow TIM_t \\ &\Box[\Box x{:}ANY]x?t \rightarrow TIM_t \\ &\Box otherwise \rightarrow TIM_{next(t)} \\ &) \end{aligned}$$

where "otherwise" is an event which is intended to occur only when nothing else can occur.

It remains to give a rigorous definition of such an event. If P is a process, we define $rescue_e(P)$ as:

$$rescue_e(P) = Q\setminus\{e\}$$

where $Q = \{s \mid s$ is in P

and if $t\langle e\rangle$ is an initial substring of s

and if $t\langle x\rangle$ is in P, then $x = e\}$

Now if the USERS are a group of processes to be executed in simulated time

simulate (USERS) = **df**

$$rescue_{timer.otherwise}\ (timer{:}TIM_o \,||\, USERS).$$

(a) Let PATHS be a set of names of unidirectional paths in a network. For each path p in PATHS:

length (p) is the time taken to traverse the path.

succ (p) is the set of paths leading from the destination of p.

$SPARK_p$ is a process representing a single traversal of path p; it is triggered by a "start" signal from one of its predecessors, and after traversing the path, it propagates a start signal to all its successors **in parallel**:

$$\begin{aligned} SPARK_p = ([\Box s{:}PATHS]s?start \rightarrow & \\ & HOLD(length(t)); \\ & ([||d{:}succ(p)]d!start \rightarrow ABORT) \\ ). & \end{aligned}$$

(b) A special path "dest" is singled out as the intended destination of a journey. It triggers the start point of the journey, and then waits for the first spark to propagate back to itself.

It then outputs the time and terminates successfully:

```
DEST = source!start;
       ([]s:PATHS]s?start →
            (timer?t:TIME → !t)
       )
```

(c) To output the length of the shortest route in the network between the source and the destination:

```
simulate ([||p:(PATHS - {dest})] p:SPARK$_p$
         || dest:DEST)
```

(d) A machine shop possesses ten groups of machines. Each group contains seven machines, which are scheduled by a foreman using a "first-come, first-served" discipline. The shop has to process a set of orders identified by names in X. Each order in turn uses a reader to input its parameters:

starttime:	at which it enters the shop,
numberofsteps:	required to fulfil the order,

and for each step:

machinegroup:	of machine needed for this step,
servicetime:	for this step.

An exclusion semaphore is required for proper sharing of the reader. Output of results has been ignored:

```
MACHINESHOP =
simulate ([rdr:MUTEX
         ||[||i:1..10]foreman$_i$:FCFS$_7$
         ||[||x:X]x:ORDER
         ])
```

Each order must read in its parameters before starting to progress in the simulation proper. All orders initially compete to use the reader for this purpose. It does not matter in what sequence they actually acquire it:

```
ORDER = [starttime: VAR$_{NN}$||numberofsteps:VAR$_{NN}$||
        rdr!acquire; (reader?n:NN → starttime := n);
        (reader?n:NN → number of steps := n);
        [[||i:1..numberofsteps]machinegroup$_i$ : VAR$_{NN}$
||[||i:1..numberofsteps]servicetime_i: VAR_NN
||(for i:1..numberofsteps →
      (reader?n:NN → machinegroup_i:= n);
      (reader?n:NN → servicetime_i:= n));
rdr!release; PROGRESS
]
```

The first action of each order is to wait until its starttime is due. It then progresses through each step, acquiring its machine from a foreman, and holding it for the required service time:

```
PROGRESS =
(starttime?m:NN → timer!n);
(for i:1..numberofsteps →
   (machinegroup_i?mg:NN →
   foreman_mg!request; foreman_mg?granted;
   (servicetime_i?n:NN → HOLD (n));
   foreman_mg!release
   )
)
```

TECHNICAL NOTES

To avoid the introduction of non-determinism, we have observed the following restrictions:

(1) Define P^o as the set of symbols denoting events in which P can participate on its first step:

$P^o = \{x \mid <x> \in P\}$

We use P□Q only when $P^o \cap Q^o = \{\}$, so that the decision between P and Q can be made on the first step.

(2) The event of ✓ occurs only at the end of a trace; and when it does occur, it is the only event that can occur:

for all traces s,

if s✓ is in P, and st is in P then t = <✓>

This ensures that successful termination of a process is always deterministic.

(3) If $\checkmark$ is in the alphabet P but not Q then P||Q is allowed only if the alphabet of Q is wholly contained in the alphabet of P. This ensures that successful termination of P automatically cuts short any further activity of Q.

(4) An alphabet transformation is always a one-one function.

(5) For s in P, P(s) describes the future behaviour of P when s is the trace of its past behaviour:

$P(s) = \{t \mid st \text{ is in } P\}$

We define $s\backslash X$ as $s\upharpoonright(\overline{P}-X)$. We insist that after concealment of X, the future behaviour of a process is still uniquely determined by the still visible symbols of its past behaviour:

For all s and t in P:

$$\text{if } s\backslash X = t\backslash X \text{ then } P(s)\backslash X = P(t)\backslash X$$

(6) An infinite array of parallel processes must not communicate with each other (their alphabets must be disjoint). This ensures that the infinite parallelism can be defined as the limit of the parallel combination of all finite subsets.

ACKNOWLEDGEMENTS

Acknowledgements are due to many colleagues. The inspiration for the design of communicating processes is due to E.W. Dijkstra. R. Milner showed how a mathematical model for them could be constructed. R. Campbell and P. Lauer are responsible for the parallel combinator. N. Francez and W.P. de Roever helped substantially in the development. J. Kennaway gave necessary mathematical support. F.M. McKeag designed the operating system and W.-H. Kaubisch designed the simulation program. Improvements over early drafts were due to perspicacious comments of E.W. Dijkstra and A. Yasuhara. The simulation example is due to O.-J. Dahl, and so is the formulation of Dijkstra's shortest path algorithm. The BUTLER is due to C. Scholten.

References

Dijkstra, E.W. (1968). Co-operating sequential processes. In _Programming Languages_, ed. F. Genuys. Academic Press, New York, pp. 43-112.

Dijkstra, E.W. (1975). Guarded commands, nondeterminacy, and a calculus for the derivation of programs. _Comm ACM_ 18.8, pp. 453-457.

Hoare, C.A.R. (1978a). Some Properties of Predicate Transformers. _JACM_ 25, 3, pp. 461-480.

Hoare, C.A.R. (1978b). Communicating Sequential Processes. _Comm. ACM_ 21, 8, pp. 666-677.

Kaubisch, W.-H., Perrott, R.H. & Hoare, C.A.R. (1976). Quasi-parallel Programming. _Software - Practice and Experience,_ 6, pp. 341-356.

Milner, Robin (1978). Synthesis of Communicating Behaviour. Mathematical Foundations of Computer Science. _Lecture Notes in Computer Science_ 64, Springer-Verlag, pp. 71-83.