My own annotated copy.

An elementary model for.

A Model for

COMMUNICATING SEQUENTIAL PROCESSES.

C.A.R. Hoare

Oxford University Computing Lbaoratory
Programming Research Group
45, Banbury Road
Oxford. OX2 6PE

Summary: A previous paper [5] 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 [6] of the sequences of possible communications between a process and its environment.

December 1978.

0. Introduction.

This paper gives a mathematical model of the concept of a communicating sequential process. A trace of the possible behaviour of a process is given by a sequence of symbols (or messages) communicated between that process and its environment. A trace may end with a special symbol ✓, indicating successful termination of the process. A trace that ends without a ✓ indicates failure or breakage of the process (e.g. abortion or deadlock). A process is defined by the set of all possible traces of its behaviour. Its actual behaviour in any given environment will be determined (at least partially) by that environment.

The set of sequences representing a process may be regarded from several different points of view. For example, the set can be identified with the (finite or infinite) tree formed by "merging" the common initial subsequences of sequences in the set.

The nodes of the tree represent states of a machine executing the process, and the arcs are labelled with the symbol whose communication (either input or output) causes the machine to pass to the state at the other end of the arc. Terminal nodes represent either success (✓) or failure (⊥). When two subtrees are equal, they can be collapsed to a single copy, with two arcs leading to its root node; and this gives yet another equivalent representation of a process as a (finite or infinite) directed graph, known as a state transition diagram.

But the most effective way of defining a process is that suggested by the work of Milner [7]; all that is necessary is to define the set of symbols which the process is capable of communicating on its first step; and then for each such symbol, to define the subsequent behaviour of the process after communication of that symbol. Of course, if a process has already successfully terminated, or is already broken, clearly there is nothing it can communicate on its first step.

These various equivalent models are described more formally in section 1 of the paper. Section 2 defines the four basic methods of constructing more complex processes out of simpler ones. They are the familiar sequential composition (concatenation), and alternative composition (modelled on guarded commands [2]); and the less familiar parallel composition, and process naming. Parallel composition is modelled by a form of interleaving, in which communications between the constituent processes are represented only once. Process naming merely ensures that every symbol communicated by the process is tagged with the process name. There is no need to introduce a special repetitive construct, since in a mathematical model, recursion is perfectly satisfactory for this purpose.

Section 3 applies the definitions given earlier to a simplified version of the language of Communicating Sequential Processes, given in [5]. It can be omitted by a reader not familiar with the previous paper.

Finally, section 4 discusses some of the defects of the model and points to possible directions of future research.

The paper is illustrated by a number of examples defining familiar concepts and algorithms. As the exposition unfolds, the examples gradually come to resemble computer programs, and the constituent processes which

enter into the definitions seem to acquire a life of their own as independent agents. From a formal standpoint, this is pure fancy; but it gives an intuitive content to the theory, and may aid our understanding and invention. I am rather pleased by the way in which the "programming language" emerges directly from the mathematical definition of its semantics; perhaps this progression should be adopted more widely as a methodology of programming language design.

1. Basic Definitions.

A process P communicates with its environment by means of symbols from its alphabet $\Sigma(P)$*. A process with alphabet Σ can terminate either sucessfully (indicated by $\surd_\Sigma$) or unsucessfully (indicated by $\perp_\Sigma$).

If a process P has not terminated, it is prepared to communicate any symbol from some nonempty set P^o contained in $\Sigma(P)$; after communication of some symbol σ in P^o, P will then pass to its next state, in which its subsequent behaviour $P(\sigma)$ in general depends functionally on σ.

More formally, the space $\mathbb{D}_\Sigma$ of processes over alphabet Σ is specified by the following axioms and definitions.

(1) $\surd_\Sigma$ is in $\mathbb{D}_\Sigma$

(2) If T is a subset of Σ and P is a total function from T to $\mathbb{D}_\Sigma$, then P is in $\mathbb{D}_\Sigma$. Such a function will be specified: $(\sigma: T \to P(\sigma))$, omitting the conventional lambda before the bound variable σ.

(3) The "broken" process is just a function with empty domain

$\perp_\Sigma =_{df} (\sigma:\{\} \to \ldots)$

(where ... could be replaced by anything, e.g. $\surd_\Sigma$)

(4) The domain P^o of a process P is defined

$(\surd_\Sigma)^o = \{\}$ (the empty set)

$(\sigma:T \to P(\sigma))^o = T$

This is the set of symbols that can be communicated by P on its first step. Clearly $P^o \subseteq \Sigma$ and $(\perp_\Sigma)^o = \{\}$

(5) If $P \in \mathbb{D}_\Sigma$ then $\Sigma(P) = \Sigma$

---

* In practice, $\Sigma(P)$ will be finite or denumerable.

(6) For all $P \in D_\Sigma$

$$P = \surd_\Sigma \quad \text{or} \quad P = (\sigma : P^0 \to P(\sigma))$$

This axiom excludes unwanted elements from D_Σ

If the domain of a function is finite, it is more convenient to define the function by simple enumeration of its arguments paired with their corresponding values. Thus if "a" and "b" are distinct symbols,

$$(a \to A \sqcup b \to B)$$

is the finite function with domain {a,b} which maps a onto A and b onto B, i.e.

$$(\sigma : \{a,b\} \to \underline{\text{if}}\ \sigma = a\ \underline{\text{then}}\ A\ \underline{\text{else}}\ B)$$

This notation is also used for finite functions with larger domains:

$$(a \to A \sqcup b \to B \sqcup \ \ldots \sqcup c \to C);$$

Of course, the symbols a,b, ..,c must all be distinct.

Example (1) X0

A process X0 is initially capable of communicating either "a" or "b", where the selection between them will be made by the environment of the process. If "a" is selected, the process will break; but if "b" is selected, the process will go into a state in which it is prepared to communicate only "a"; if and when this is accomplished, the process terminates successfully. Such a process can be defined:

$$X0 = \text{df}\ (a \to \bot \sqcup b \to (a \to \surd))$$

$p?x : \{0,1\} \to \underline{\text{if}}\ x = 0\ \underline{\text{then}}$ abort $\underline{\text{else}}$ $q!1$

The explanation given above carefully avoids ascribing any direction to the communication between a process and its environment. This strange assimilation of input with output is a purely formal simplification. Informally, when a process can communicate only a single symbol (e.g., on the second step of X0), we shall call this "output"; but if several alternatives are offered (as on the first step of X0), we shall regard the process as asking for "input" of information from its environment.

In future, output will be denoted simply by writing the character to be ~~ouput~~ communicated:

i.e. "a" for "$(\sigma : \{a\} \to \surd)$"

Thus X0 could be written more briefly as

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

Similarly, the theory gives no particular interpretation to the symbols in the alphabet of a process. Sometimes they are just symbols of a language to be accepted by an automaton. Sometimes they are

"commands" or "questions" from one process to another, and sometimes they are "responses" or "replies". Such different interpretations of the symbols will feature in the examples given throughout this paper.

An equivalent way of defining a process P is in terms of its language $\mathcal{L}(P)$. This is the set of finite sequences of symbols which it can communicate during some finite interval of time. By this definition, if s is in $\mathcal{L}(P)$, so is every initial subsequence of s.

The symbol ✓ is reserved to appear only at the end of sequences which lead to successful termination. Any other sequence, which is not an initial subsequence of some longer sequence of the language, leads to breakage of the process. More formally, the language of a process is defined

$$\mathcal{L}(P) = \{\varepsilon, \checkmark\varepsilon\} \quad \text{if } P = \checkmark$$

$$= \{\varepsilon\} \cup \{\sigma\, s \mid \sigma \in P^{o} \;\&\; s \in \mathcal{L}(P(\sigma))\} \quad \text{otherwise}$$

where ε is the empty sequence and σs is formed by prefixing the symbol σ to the sequence s, (so that $\checkmark\varepsilon$ is the sequence containing only ✓).

From this definition it follows that:

$\mathcal{L}(\perp) = \{\varepsilon\}$

$\mathcal{L}(a) = \{\varepsilon, a, a\checkmark\}$

$\mathcal{L}(a \to A \sqcup b \to B) =$

$\{\varepsilon\} \cup \{as \mid s \in A\} \cup \{bs \mid s \in B\}$

$\mathcal{L}(X0) = \{\varepsilon, a, b, ba, ba\checkmark\}$

The appending of ✓ to mark the end of a successful trace will turn out to be a convenient coding convention.

It is sometimes illuminating to picture the behaviour of a process as a tree. The nodes of the tree represent states of the process, the initial state being at the root of the tree. Each branch leading from a node is labelled by a distinct symbol from the alphabet of the process; communication of that symbol causes the process to pass along the branch which it labels. A terminal node (leaf) of the tree is marked with a ✓ to indicate successful termination; a leaf which is not so marked indicates breakage (⊥). For example, the process X0 defined above is shown in Fig 1.

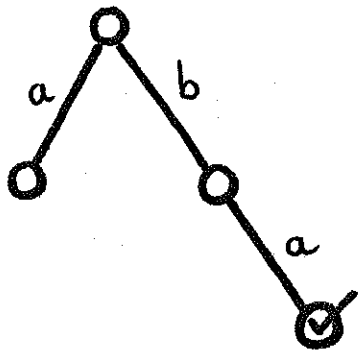

Figure 1.

The language corresponding to such a tree contains the sequence of labels on every finite subpath which starts at the root; and a final ✓ is recorded at the end of each path which leads to successful termination.

Such a tree can be drawn for any process which has only a finite language; but for an infinite language, informal dots would be required. For example, a process X1 which communicates a string of none or more "c"s followed by a single "d" can be pictured as in Fig 2 (a).

In many cases such an infinite tree can be represented as a directed graph (Fig. 2(b)).

But such a graph must be regarded as equivalent to one with its cycles arbitrarily expanded, so that (a) (b) and (c) of figure 2 all depict the same process.

In functional notation, such an infinite tree has a simple recursive definition:

$$X1 =_{df} (c \rightarrow X1 \sqcup d \rightarrow \surd)$$

However, in other cases (e.g. Fig 5), a process defined by recursion cannot be represented as a finite graph.

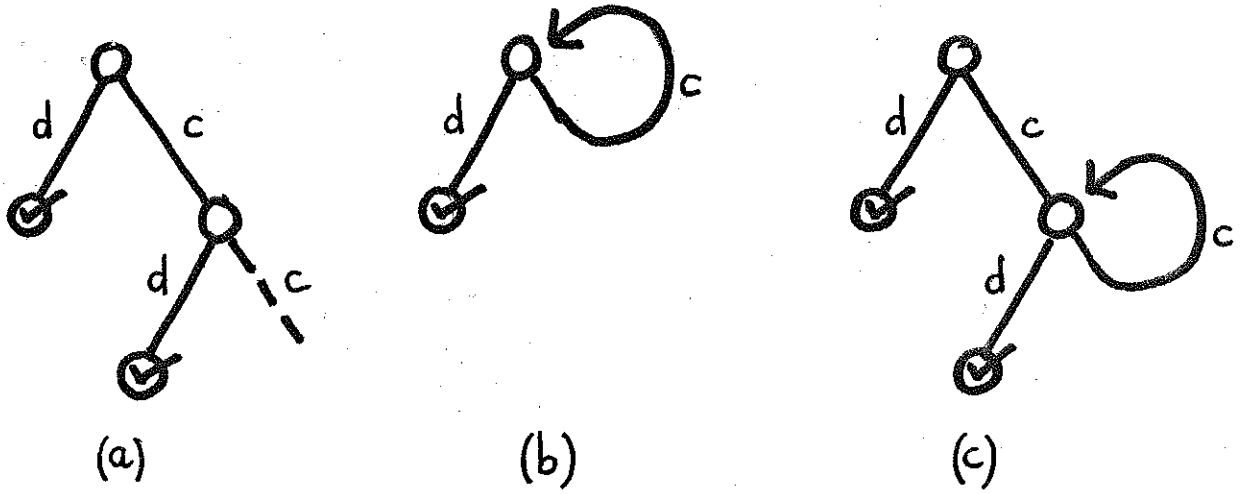

Figure 2.

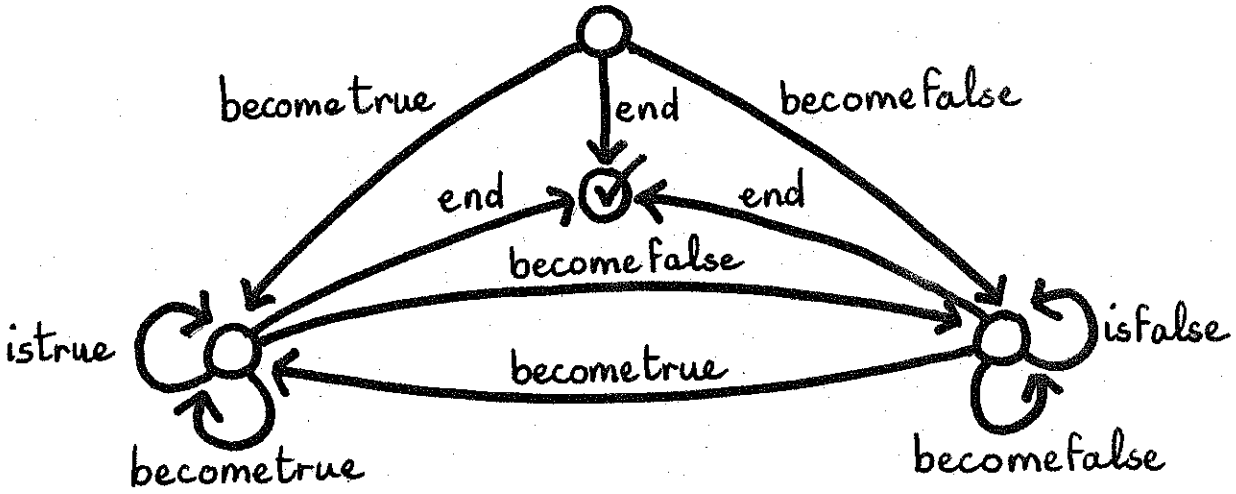

Figure 3.

Example: (2) BOOL

A process BOOL models the behaviour of a Boolian variable. Commands and questions from its environment are represented by symbols in its alphabet. At any time it accepts commands "become true","become false" and "end". After receiving "end", it terminates successfully. When true it accepts the symbol "istrue" and when false it accepts the symbol "isfalse", but initially it accepts neither of these.

BOOL $=_{df}$ p?(become true → T
□ become false → F
□ end → ✓
)

where T behaves like a true Boolean and F like a false one:

T $=_{df}$ p?(become true → T
□ become false → F
□ ~~is true~~ → T
□ end → ✓
)

F $=_{df}$ p?(become true → T
□ become false → F
□ ~~is false~~ → F
□ end → ✓
)

This process is pictured as a directed graph in Fig 3.

Technical Note.

The use of recursion to define potentially infinite processes requires justification in the form of complete partial ordering($\underline{\subseteq}$) over processes. The simplest candidate for such an ordering is:

$$P1 \underline{\subseteq} P2 = \text{df } \mathcal{L}(P1) \underline{c} \mathcal{L}(P2)$$

The concerned reader should check that all the operators defined in the next section are continuous with respect to this ordering. However, there are other orderings which preserve continuity of the operators; and for some purposes (e.g. the definition of simulated time, Section 4.4) one of these may be preferable.

2. Composition of ~~Deterministic~~ Processes.

In this section we introduce various methods of defining compound processes from simpler component processes. Three operators will be introduced, and their syntactic precedence will be as listed:

;	sequential composition (binds tightest)
||	parallel composition
⫅	alternative composition.

~~The infix dot, used for compound symbols and process naming, binds tighter than all of these.~~ However, redundant brackets will be used quite freely.

2.1. Sequential Composition.

A sequential composition (A;B) is a process which behaves like A until A terminates. If A has broken, then so has (A;B); otherwise, its subsequent behaviour is like B. More formally

$$\Sigma(A;B) = df \quad \Sigma(A) \cup \Sigma(B)$$

$$A;B \quad = df \quad B \qquad \text{if } A = \surd$$

$$= (\sigma: A^{o} \to (A(\sigma);B)) \qquad \text{otherwise}$$

Clearly, this operation is associative, and

$$A;\surd = \surd;A = A \quad \text{(skip)}$$

$$\perp;A = \perp \quad \text{(abort)}$$

$$(a \to A \between b \to B);C = (a \to (A;C) \between b \to (B;C))$$

$$(p?x:T \to f(x));B = (p?x:T \to (f(x);B))$$

Sequential composition may be pictured as an operation on trees, which grafts the whole of the tree B in place of every ticked leaf of A (Fig. 4).

$$A;B = \{st \mid s \in A \ \&\ t \in B \ \&\ (s\surd \in A \vee t = \varepsilon)$$
$$s \text{ does not end in } \surd\}$$

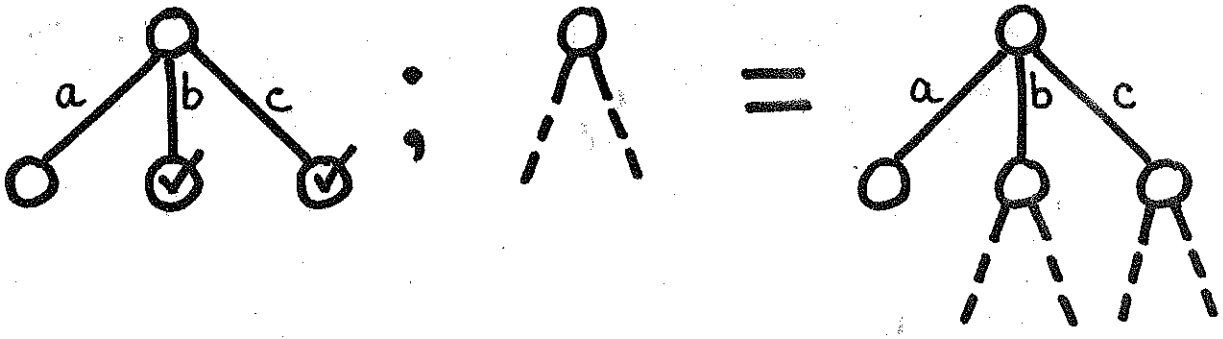

Figure 4.

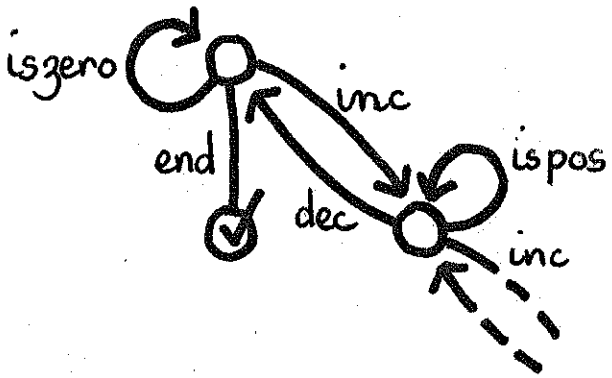

Figure 5

Note that unticked leaves of the first operand represent failure, and remain ungrafted.

The language associated with composition is

$\mathcal{L}(A;B) = \{s;t \mid s \in A \,\&\, t \in B\}$

where $s;t = s$ if s does not end in $\surd$

and $s\surd;t = st$ (i.e. concatenation of s and t).

Examples.

(1) The process X2 accepts strings of the form $a^n b^{n+1}\surd$.

X2 = ρ?(a → (X2;b) ⊔ b → ρ)

(2) The process X3 accepts strings of the form $a^n b^n c^m d\surd$

X3 = ρ?(a → (X2;X1) ⊔ c → X1 ⊔ d → $\surd$)

where X2 and X1 have been defined above

(3) The process X4 accepts strings of the form $b^n c^n d\surd$
[left as an exercise]

(4) The process CTR behaves like a simple integer counter initialised to zero. The symbols of its alphabet represent commands and questions from its environment; on receiving a command "inc", its value increases by one; on receiving "dec", its value (provided it is non zero) decreases by one. A zero counter can accept the command "end", after which it terminates successfully. When its value is positive it will accept the symbol "ispos", and when it is zero, it will accept "iszero".

CTR = df ?(is zero → CTR
⊔ inc → (POS;CTR)
⊔ end → $\surd$
)

where POS is designed to terminate after accepting one more "dec" than "inc":

POS = ?(ispos → POS
⊔ dec → $\surd$
⊔ inc → (POS;POS)
)

The behaviour of CTR is pictured as an infinite graph in Fig 5.

2.2. Parallel Composition.

A parallel composition (A||B) is a process whose communications are a kind of merging of the individual communications of A and B. More precisely, any symbol σ which belongs only to the alphabet of A (i.e. $\sigma \in \Sigma(A) - \Sigma(B)$) will be communicated directly between the environment and A; and similarly, a symbol $\sigma \in \Sigma(B) - \Sigma(A)$ will be accepted by A||B only if and when it is accepted by B. But a symbol σ which is in both their alphabets must be accepted simultaneously by both A and B. (A||B) terminates successfully only if and when both A and B have terminated successfully.

More formally, the language of A || B can be defined as follows:

$$\Sigma (A \,||\, B) = \Sigma(A) \cup \Sigma(B)$$

$$\mathcal{L}(A||B) = \{s \mid (s \restriction \Sigma(A)) \in \mathcal{L}(A) \;\&\; (s \restriction \Sigma(B)) \in \mathcal{L}(B)\}$$

where $s \restriction \Sigma$ is the string formed from s by ~~omitting all~~ including only symbols in ~~outside~~ $(\Sigma \cup \{\surd\})$, and omitting all others.
The ||operator is associative & commutative. It was discovered independently by Francez and by Plotkin.

Example.

If $\mathcal{L}(X0) = \{\varepsilon, a, b, ba, ba\surd\}$ where $\Sigma(X0) = \{a,b\}$
and $\mathcal{L}(B) = \{\varepsilon, b, bc, bcb, bcb\surd, bd, bda, bda\surd\}$
where $\Sigma(B) = \{a,b,c,d\}$

then $\mathcal{L}$ (X0||B) = $\{\varepsilon, b, bc, bd, bda, bda\surd\}$

This example is pictured in figure 6.

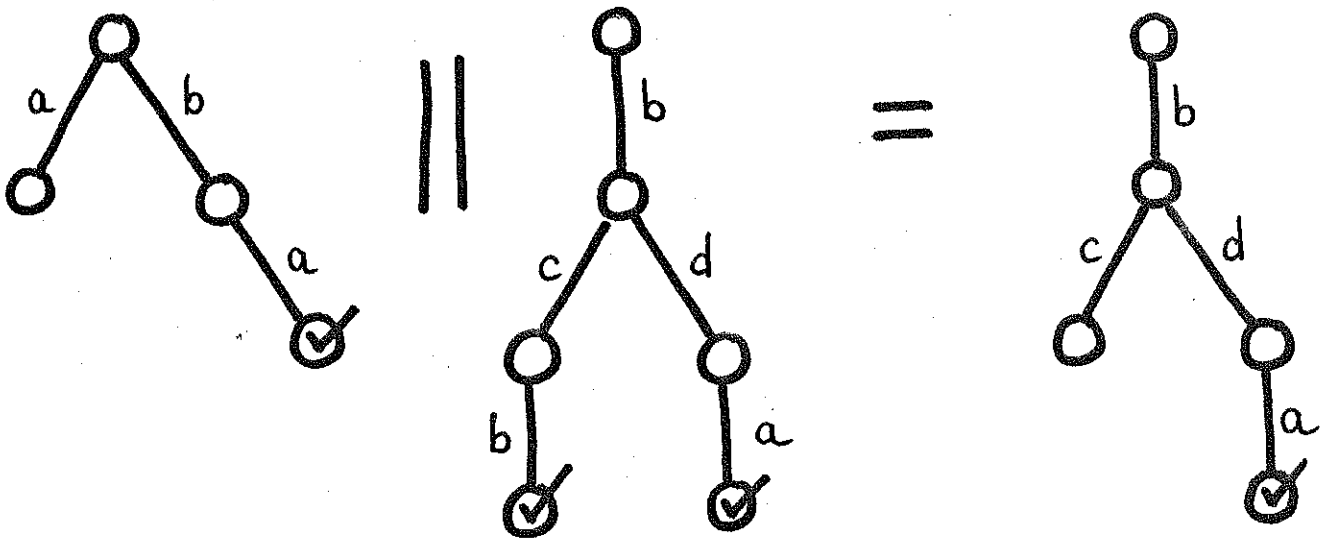

Figure 6

Note how the initial "output" of "b" by B has been "input" by X0, and used to steer X0 away from its failing left branch. After communication of "b", X0 has to wait until B has input from their common environment either the symbol "c" or the symbol "d", which are of no concern to X0. If "d" is input, then X0 and B agree that the next symbol must be "a", which is in both their alphabets. But if "c" is input, X0 will accept only "a" and B will accept only "b". Since these two symbols are in both of their alphabets, the disagreement about their acceptability leads to failure of (X0||B) in a kind of deadly embrace. [1]

The general definition of parallel composition describes an unfamiliar and slightly complex combination of interleaving and intersection of the communications of the component processes. However, there are two common special cases, in which the aspects of interleaving and intersection are isolated:

(1) Intersection.

If $\Sigma(A) = \Sigma(B)$ then <u>every</u> symbol communicated by A||B must be simultaneously communicated by both A and B; and consequently

$$\mathcal{L}(A||B) = \mathcal{L}(A) \cap \mathcal{L}(B)$$

(2) Interleaving

If $\Sigma(A) \cap \Sigma(B) = \{\}$ then <u>every</u> symbol communicated by A||B must be communicated either by A or by B, but <u>not both</u>. Thus $\mathcal{L}(A||B)$ contains the set of all interleavings of an arbitrary unticked string $a \in A$ and an arbitrary unticked string $b \in B$; The interleaving is ticked in $\mathcal{L}(A||B)$ only if both $a\surd \in \mathcal{L}(A)$ and $b\surd \in \mathcal{L}(B)$.

The direct definition of the parallel operator in terms of the functions rather than the languages is rather more complex:

i $A||B = \surd$ if $A = B = \surd$

ii $(A||B)^0 = (A^0 \cap B^0) \cup (A^0 - \Sigma(B)) \cup (B^0 - \Sigma(A))$

iii $(A||B)(\sigma) = A(\sigma)||B(\sigma)$ if $\sigma \in \Sigma(A) \cap \Sigma(B)$

$= A(\sigma)||B$ if ~~$\sigma \in \Sigma(B)$~~ $\sigma \in \Sigma(A) - \Sigma(B)$

$= A||B(\sigma)$ if ~~$\sigma \in \Sigma(A)$~~ $\sigma \in \Sigma(B) - \Sigma(A)$

This definition makes it clear that each symbol accepted by A||B must be accepted by each process which has that symbol in its alphabet.

Examples.

(1) The process X5 communicates strings of the form $a^n\ b^n\ c^n\ d\surd$

$X5 = X4||X3$

where X4 and X3 were defined in the previous section, examples (2) and (3).

(2) The process X6 uses the process CTR to count up to three and down again.

X6 =inc; inc; inc; DWN

where DWN = (ispos → dec; DWN
⊔is zero → end)

It runs in parallel with CTR, thus: CTR||X6

ℒ(CTR||X6) consists of all initial substrings of:
inc inc inc ispos dec ispos dec ispos dec iszero end✓

(3) The process ACTR behaves exactly like CTR except that it communicates symbols ("a.inc", "a.dec", etc., instead of "inc", "dec", etc. Symbols with an infix dot are known as compound symbols

ACTR = (a.iszero→ACTR

⊔a.inc →APOS; ACTR

⊔a.end → ✓

)

APOS = (a.ispos → APOS

⊔a.dec → ✓

⊔a.pos → APOS; APOS)

(4) The process ABCTR behaves like a pair of counters, one of which accepts commands of the form a.inc, a.dec, etc., and the other accepts commands of the form b.inc, b.dec, etc:

ABCTR = df ACTR||BCTR

where the definition of BCTR is very similar to that of ACTR. Because their alphabets are disjoint, ACTR and BCTR operate quite independently of each other.

(5) The process ABCTR is to run in parallel with a USER process, (still to be determined) thus:

ABCTR||USER

As part of the USER process, we require a process ADD which will add the current value of BCTR to ACTR:

ADD = ~~df~~ (b.iszero → ✓

⊔b.ispos → a.inc;

b.dec; ADD;b.inc

)

2.3. Process Naming.

The examples of the previous section show the usefulness of compound symbols, such as "a.inc" and "b.dec", in distinguishing between different processes with very similar behaviour. The first component of the symbol, the "a" or the "b", serves as the name for a process which communicates the symbol. This permits the environment of a group of parallel processes to direct a communication to the particular one of them which answers to that name.

More formally, if "x" is a symbol and "a" is a simple symbol, then "a.x" is also a symbol; and "a.x" = "b.y" only if "a" = "b" and "x" = "y". The dot - prefixing notation can be extended to alphabets and to processes:

$$a.\Pi = df\ \{a.\sigma \mid \sigma \in \Pi\}$$
$$\Sigma(a.A) = df\ a.\Sigma(A)$$
$$a.A = df\ \checkmark \quad \text{if } A = \checkmark$$
$$= (\sigma : (a.A^{o}) \not\to a.(A(\delta))) \quad \text{otherwise,}$$

where δ is formed from σ by removing its intitial "a.".
In order to avoid the awkwardness of this last construction, we introduce the query notation:

$$a?\delta : \Pi \to P(\delta) \quad \text{for} \quad \sigma : (a.\Pi) \to P(\delta)$$

which "inputs" any symbol δ in Π which has "a" as prefix.
The process naming operation has obvious properties

$$a.\checkmark = \checkmark$$

$$a.\perp_{\Sigma} = \perp_{a.\Sigma}$$

$$a.(A;B) = (a.A);(a.B)$$

$$a.(A||B) = (a.A)||(a.B)$$

$$\mathcal{L}(a.A) = \{a.s \mid s \in \mathcal{L}(A)\}$$

where a.s is formed from s by prefixing "a." to each symbol of s except $\checkmark$.

Examples.

(1) The following is a more convenient definition of ABCTR

$$ABCTR =_{df} a.\ CTR||b.CTR$$

(2) A process FAC inputs a natural number $n \in NN$ and later outputs the value of its factorial. When $n>0$, a new process f.FAC is "created" to compute the value of factorial (n - 1).

$$FAC =_{df} (n{:}NN \to \underline{\text{if}}\ n = 0\ \underline{\text{then}}\ 1\ \underline{\text{else}}\ (f.FAC||X))$$

where $X =_{df} (f.(n-1);\ (f?m{:}\ NN \to m \times n))$

Note that $\Sigma(X) = \bigcup_{n \in NN} \{n, f.n\}$

$$\Sigma(f.FAC) = \bigcup_{n \in NN} \{f.n, f.f.n, f.f.f.n \ldots\}$$

$$\Sigma(FAC) = W\,\Sigma(X) \cup \Sigma(f.FAC) = NN \cup f.\Sigma(FAC)$$

Thus in their parallel composition $(f.FAC||X)$, the output of the natural number m x n by X is of no concern to f.FAC; and the internal communications between subprocesses of f.FAC are of no concern to X. The number of "f."s in a symbol indicates the depth of recursion at which it is communicated. Some strings of $\mathcal{L}(FAC)$ are:

01✓

1 f.0 f.1 1✓

2 f.1 f.f.0 f.f.1 f.1 2✓

3 f.2 f.f.1 f.f.f.0 f.f.f.1 f.f.1 f.2 6✓

The presence in these strings of symbols with the form f.n, f.f.n, etc. is very inconvenient, and it would be nice to remove them. Unfortunately, ~~in general the omission of symbols will introduce nondeterminism, and we must postpone the solution of this problem to a later section.~~ this is not easy in the general case. The topic is postponed to Section ~~4.2~~ 2.5.

In view of the possible novelty of the example, it is worth recalling that the recursion is interpreted as the limit of an ascending chain of sets

$$\mathcal{L}(FAC) = \bigcup_{i \in NN} \mathcal{L}(FAC_i)$$

where $FAC_0 = \perp_{\Sigma(FAC)}$

$$FAC_{i+1} = (n{:}NN \to \underline{\text{if}}\ n{=}0\ \underline{\text{then}}\ 1\ \underline{\text{else}}\ (f.FAC_i||X))$$

FAC_i will successfully compute the factorial of any number less than i; and will fail for any larger number.

The suspicious reader may attempt to draw outline tree pictures of X_*, FAC_1, $f.FAC_1||X_*$, FAC_2, and FAC_3. But pictures are a very weak aid to the understanding of algorithms, even one as simple as this.

2.4 Alternative Composition.

If A and B are processes not equal to $\surd$, and there is no symbol acceptable on the first step by both A and B, then $A \sqcup B$ is the process which initially accepts any symbol acceptable to either A or B; if the symbol is actually acceptable by A, the subsequent behaviour of $A \sqcup B$ is determined by A; or if it is acceptable by B, ~~it is B that determines it.~~ subsequent behaviour is determined by B.

More formally, if $A \neq \surd$ and $B \neq \surd$ and $A^o \cap B^o = \{\}$

$$\Sigma(A \sqcup B) = \Sigma(A) \cup \Sigma(B)$$

$$A \sqcup B =_{df} (\sigma : (A^o \cup B^o) \rightarrow \underline{\text{if}}\ \sigma \in A^o\ \underline{\text{then}}\ A(\sigma)\ \underline{\text{else}}\ B(\sigma))$$

This is, of course, the familiar union of functions with disjoint domains, and is associative, commutative and idempotent. Further:

$$(A \sqcup B);C = (A;C) \sqcup (B;C)$$

and

$$\mathcal{L}(A \sqcup B) = \mathcal{L}(A) \cup \mathcal{L}(B) \quad \text{provided that } \mathcal{L}(A) \cap \mathcal{L}(B) = \{\varepsilon\}$$

In applying this notation, we shall abbreviate "$\sigma : \{a\} \rightarrow A$" to "$a \rightarrow A$".

Thus the familiar construct "$(a \rightarrow A \sqcup b \rightarrow B)$" is just a special case of alternative composition, where the domain of each component is a singleton set.

Examples.

(1) A process COPY inputs natural numbers $n \in NN$ from a process named "west", and outputs them to a process named "east". On receiving an "end" signal, it transmits "end" to the east, and terminates successfully.

```
COPY = (west?n:NN → east.n; COPY
       ⊔ west.end → east.end
       )
```

(2) A process REGNN models the behaviour of a register, initially empty, but capable of containing a single natural number. It responds to a command "assign.n" where $n \in NN$. It also accepts the symbol "fetch.m" where m is the number most recently assigned to it. It terminates on receipt of an "end" command. (Note that in this case the compound symbols indicate the nature of the command, not the name of its source)

prefix.

REGNN = (assign?n:NN → X_n ⊔ end → ✓)

where for n ∈ NN

X_n = (fetch.n → X_n

⊔ end → ✓

⊔ assign?m:NN → X_m

)

This is a form of mutual recursion with an infinite number of simultaneous definitions of X_i for $i = 0,1,2,...$ The definition is theoretically sound, but it may be disturbing to those not familiar with the technique.

(3) A process NNSET is required to model a set of natural numbers, initially empty. It accepts commands "insert.n", and inserts the number n into the set. It responds to commands "has.n" by outputting "yes" if n has been previously inserted into the set, and "no" otherwise. It terminates successfully on receiving the symbol "end".

NNSET = (has?n:NN → no

⊔ insert?n:NN → (rest.NNSET||X_n)

⊔ end → ✓

)

where

X_n = (has?m:NN →

(if m = n then yes

else rest.has.m;

(rest.yes → yes ⊔ rest.no → no)

) ; X_n

⊔ insert?m:NN →

if m = n then X_n

else (rest.insert.m;X_n)

⊔ end → rest.end

)

Some strings of £(NNSET) are:

end ✓

has.3 no end ✓

insert.3 has.3 yes

insert.3 has.4 rest.has.4 rest.no no

insert.3 insert.4 rest.insert.4 has.4 rest.has.4 rest.yes yes

3. Communicating Sequential Processes.

In this section, we define a slightly simplified version of the language of Communicating Sequential Processes [5], and illustrate it by the example of the dining philosophers.

3.1 Output commands.

The basic program structure of [5] is the parallel command, consisting of a fixed number of named processes. This can be modelled by process naming, thus:

$$(a_1.A_1 || a_2.A_2 || \ldots || a_n.A_n)$$

where $a_1, a_2, \ldots, a_n$ are distinct process names

and $A_1, A_2, \ldots, A_n$ describe the behaviour of the component processes.

But in this construction, the alphabet of each component process is prefixed by its name, and is therefore disjoint from the alphabet of every other process. Consequently, we must introduce a special convention to enable the processes to communicate and synchronise with each other.

This is accomplished by introduction of a new kind of compound symbol, "a!σ", where a is a process name, and σ is a symbol. When this symbol is output by a process with name b, the symbol actually transmitted is prefixed by b: "b.(a!σ)". But we declare that this double compound symbol is identically equal to the symbol with the process names reversed: "a.(b.σ)". This symbol will in general be in the alphabet of a process with name "a", and can be input by an input command (within that process) thus:

(b?x:Π →...x...).

This convention has the realistic consequence that each act of communication (or synchronisation) involves exactly two processes, exactly one of which (the source) must issue an "output command" of the form "a!σ", where "a" is the name of the destination of the output.

Unfortunately, this technique, which works well for internal communication between processes of a parallel command, does not work so well for communication between the individual processes and their global environment. The reason is that each such communication is still tagged with the local process name of the participating process. The removal of these local names is a topic that will be treated in section 4.2.

Example: dining philosophers.

This example is the same as example 5.3 of [5]. It uses arrays of processes, which can be simply defined in an informal manner:

$$||_{i:m..n} P_i = \text{df.}\ P_m || P_{m+1} || \ldots || P_n$$

and $$\sqcup_{i:m..m} P_i = \text{df}\ P_m \sqcup P_{m+1} \sqcup \ldots \sqcup P_n$$

The solution is:

$$(room.ROOM_4 || (||_{i:0..4} fork_i.FORK_i) || (||_{i:0..4} phil_i.PHIL_i))$$

where $PHIL_i$ = THINK;
(end → ✓

⊔ room!enter →

fork$_i$!pickup; fork$_{i\oplus 1}$!pickup;

EAT;

fork$_i$!putdown; fork$_{i\oplus 1}$!putdown;

room!exit; $PHIL_i$

)

and $FORK_i$ = (end → ✓

⊔ phil$_i$.pickup → phil$_i$.putdown; $FORK_i$

⊔ phil$_{i\ominus 1}$.pickup → phil$_{i\ominus 1}$.putdown; $FORK_i$

)

and $ROOM_0$ = ($\sqcup_{i:0..4}$ phil$_i$.enter → $ROOM_1$

⊔ end → ✓

)

$ROOM_4$ = ($\sqcup_{i:0..4}$ phil$_i$.exit → $ROOM_3$)

$ROOM_n$ = (($\sqcup_{i:0..4}$ phil$_i$.enter → $ROOM_{n+1}$)

⊔ ($\sqcup_{i:0..4}$ phil$_i$.exit → $ROOM_{n-1}$

) for $0<n<4$.

$ROOM_n$ describes the behaviour of the room with n philosophers inside it. The restriction of this number to four prevents deadlock.

3.2 Guarded Commands.

It is common that selection between alternative commands needs to be determined by the truth or falsity of a Boolean condition, rather than by input from an environment. If we confine attention to the simple case where the condition is just the value of a Boolean variable, this can be easily achieved. Let b and c be distinct names of processes which model Boolean variables. (See section 1 example (2) BOOL). Then (b.istrue → B ⊔ c.istrue → C) is an alternative command which can behave like B if b is true or like C if c is true. If both are true, the choice is not determined. If neither is true, neither of the communications b.istrue or c.istrue can take place, and the alternative command fails. Of course "!isfalse" can be used in place of ".istrue" wherever negation of the condition is required.

Often, selection in an alternative command needs to be determined by a combination of circumstances, for example by a combination of truth of a condition "b" and the readiness of some other process to communicate some symbol "s". To cater for this, the paper [5] introduced multiple guards, in which an input command appearing to the left of an → could be preceded by one or more Boolean conditions, falsity of which would inhibit selection of that alternative. This effect can be achieved by the definition

$$(b.istrue;s \to A \sqcup t \to C) =_{df} (b.istrue \to (s \to A \sqcup t \to C) \sqcup t \to C).$$

The definition can be readily extended to cases where more than one of the alternatives have multiple guards, for example:

$$(b.istrue;s \to B \sqcup c.istrue;t \to C)$$

$$= b.istrue \to (c.istrue;t \to C \sqcup s \to B) \sqcup c.istrue \to (b.istrue;s \to B \sqcup t \to C)$$

The case of several Boolean conditions on a single guard can be defined similarly:

$$(b.istrue;c.istrue;s \to B \sqcup t \to C) = (b.istrue \to (c.istrue;s \to B \sqcup t \to C) \sqcup t \to C)$$

If all these definitions are fully expanded, the number of cases grows exponentially.

The complexity of the construction would be uncontrollable if it were not guaranteed that the tests on the preliminary guards "b.istrue" "c.istrue" are free of side-effect.

Example: ROOM

This example redefines the behaviour of the ROOM in which the philosophers eat. It uses an integer register "occ" instead of the parameter.

```
ROOM =df  (occ.REGNN || full.BOOL ||
          (occ.assign0 || full.becomefalse);
          LOOP;
          occ!end ; full!end
          )

where LOOP = (⊔ i.0..4 full.isfalse;phil_i.enter →
          (occ.fetch?n: {0..2} → occ.assign.(n+1);
           ⊔ occ.fetch.3 → occ.assign.4;
                          full.becometrue
          ) ); LOOP
          ⊔(⊔ i:0..4 phil_i.exit →
                (occ.fetch?n:{1..4} →
                     occ.assign(n-1);
                          full.becomefalse
          );LOOP
          ⊔ end → ✓
          )
```

3.3. Automatic termination.

In [5] it was postulated that a repetitive command with input guards would automatically terminate if the source of every input guard had terminated. This convention is quite convenient to use; but the construction of a mathematical model involves the same kind of complexity that would be necessary in a practical implementation on one or more computers. We arrange that before a process "a" terminates it is prepared to communicate the symbol "a.ended" as often as is required; but that when all processes of a parallel command have terminated, they all simultaneously communicate the symbol "finished" (which is in the alphabet of all of them). This is achieved by appending after the process an endloop of the form

```
ENDLOOP (a) = (σ:PROC.a.ended → ENDLOOP(a)
              ⊔ finished → ✓
              )
```

where PROC is the set of all ~~simple and compound~~ process names in the same parallel command.

and PROC.a.ended = {π.a.ended | $\pi \in$ PROC}

~~The introduction of the set PROC is required to enable the test "a.ended" to be made at any depth of process nesting and naming.~~ Thus we can define the double colon notation of [5]:

$$a::A =_{df} (a.A); \text{ENDLOOP}(a).$$

Now, within another process B we can at any time test whether the process a::A has ended; and in particular, this information can be used to terminate a loop. Thus we can define the one-limbed repetitive command of [5]:

$$*[a?x:T \rightarrow L] =_{df} \text{LOOP}$$

where LOOP = (a.ended $\rightarrow \checkmark$

$\square$ a?x:T $\rightarrow$ L;LOOP

)

The case of a loop with several alternatives uses the convention introduced for chains of Boolean guards, e.g.

$$*[b?x:T\,1 \rightarrow L1 \;\square\; c?x:T\,2 \rightarrow L2] =_{df} \text{LOOP}$$

where LOOP =

(b.ended ; c.ended $\rightarrow \checkmark$

$\square$ b?x:T 1 $\rightarrow$ L1;LOOP

$\square$ c?x: T2 $\rightarrow$ L2;LOOP

)

4. Discussion:

This paper has presented a rather general model of parallelism and communication, and has applied the model to the definition of a number of familiar and less familiar programming concepts. The complexity of the definition (in spite of some fairly ingenious notational conventions) reveals an unexpected logical complexity in some of the apparently primitive ideas of Communicating Sequential Processes. This section speculates on further developments and applications of the model.

4.1. Correctness.

A grave defect of this paper has been that it contains a large number of complex mathematical definitions, and makes no attempt to derive from them any useful theorems. The most useful theorems would prove the correctness of processes, with respect to some specification of their intended purpose. Clearly, the purpose of a process P is to interact successfully with some environment E. More specifically, we can define an environment E as being

itself a process with $\overline{\Sigma}(E) \subseteq \Sigma(P)$. On a given step, E is capable of communicating any symbol of E^o with P; and for each σ in E^o, the subsequent behaviour of E is defined by $E(\sigma)$. The outcome is satisfactory only if E ends in $\surd$ and P ends in $\surd$ as well. But if either P or E breaks, or if at any stage $P^o \cap E^o = \{\}$ (deadlock), then the interaction has failed. P is correct with respect to E if there is no possibility of such failure.

More formally, the interaction between a process P and its environment E is defined simply by their parallel composition P||E. If every branch of P||E is finite and ends in $\surd$, then there is no way in which their interaction could fail to terminate successfully; and we say that the process P||E is satisfactory, and that the process P is correct in environment E.

Simple consequences of this definition are

(1) $\perp$ is never correct

(2) $\surd$ is correct only in environment $\surd$

(3) If P is correct in E and Q is correct in F then (P;Q) is correct in (E;F).

It would be very desirable to derive a simple but complete calculus of correctness for communicating processes, similar to that formulated in [2] for sequential processes; but simplicity may be achievable only in carefully defined special cases.

4.2. Localisation of Internal Communication.

In general the environment E of a process P is not concerned with internal communications between named subprocesses of P; it cannot control their timing, nor can it even detect their occurrence. To model the hiding of the internal communications of P all that is necessary is to exclude the symbols used for internal communication from the alphabet of the environment E. Thus, whenever there is an option of an internal communication, that option can be exercised by P alone, without participation of E. If there are several options, or if there is a choice between internal and external communication, the choice is non-deterministic. If P is correct with respect to E, then their interaction will terminate successfully, in spite of the nondeterminism. Note that if there is a possibility of an infinite sequence of internal communications, unpunctuated by an external one, then, by our definition, P is not correct in environment E, because P||E will contain an infinite branch.

This technique can be taken as the basis of a method for localisation of internal communications. Let P be a process, and let π be the set of symbols used for internal communication within P. We wish to define an operation $P \setminus \pi$, with alphabet $(\Sigma(P) - \pi)$, which behaves exactly like P, except that the occurrence of all communication of symbols from π has been wholly concealed. Clearly, we want to ensure that $P \setminus \pi$ behaves exactly like P in all environments E with alphabets which exclude π, i.e.

$$\forall E \; \Sigma(E) = \Sigma(P) - \pi \Rightarrow$$

$$P \text{ is correct in } E \equiv (P \setminus \pi) \text{ is correct in } E.$$

In general, there is more than one $P \setminus \pi$ which has this property; and the choice of which one to use can be left non-deterministic. Unfortunately,

treatment of this form of nondeterminism is beyond the scope of the simple model presented here.

4.3. Analysis of Algorithms.

If P is correct in environment E, it is reasonable to ask the additional question, "How soon can one be certain that their interaction will terminate?" Let $c(\sigma)$ be the time taken to communicate the symbol σ, for all $\sigma \in \Sigma(P)$. Consider a particular trace $s\surd$ from $\mathcal{L}(P||E)$. Define $\$(s) = \sum_i c(s_i)$. This is the time taken by trace s on the worst-case assumption that only one communication can take place at a time, as would occur if P were executed on a single processor which simulates its parallelism by timesharing between processes. In such a case, the answer to the question posed above is

$$\max \{\$(s) \mid s\surd \in \mathcal{L}(P||E)\}$$

Now consider the best case assumption, in which a process is executed with as much parallelism as possible. Two communications can occur in parallel if and only if their sources and destinations are all distinct process names.

Let s be the trace which we are trying to analyse. Let $T_s(P)$ be the earliest time at which process named P will terminate, when the trace of the overall behaviour of the system is s. Clearly $T_{\varepsilon}(P) = 0$ for all p. Extension of s by a communication σ, with source a and destination b, will clearly have to be delayed until both a and b are free; and then it will occupy both a and b for a further $c(\sigma)$ units of time. Thus

$$T_{s\sigma}(P) = T_s(P) \qquad \text{if sourse } (\sigma) \neq P \neq \text{dest}(\sigma)$$

$$T_{s\sigma}(\text{source}(\sigma)) = T_{s\sigma}(\text{dest}(\sigma)) =$$

$$c(\sigma) + \max \{T_s(\text{source}(\sigma)), T_s(\text{dest}(\sigma))\}$$

The time when all processes are finished is

$$\$(s) = \max \{T_s(P) \mid P \in PROC\}$$

where PROC is the set of all process names.

This definition works only if the participants in each communication can be identified.

4.4. Discrete event simulation.

The theory described so far in this paper places no constraint on the manner in which parallelism is implemented; it can be applied equally to multiprocessor networks, and to quasiparallel implementations on a single processor; consequently, the execution of a group of processes in quasiparallel can serve as a faithful simulation of a genuinely parallel system. The usefulness of such simulations can be greatly increased if it is possible to specify explicitly the passage of simulated time, and dissociate this completely from the passage of time on the processor conducting the simulation.

In a simulation language like SIMONE [4] this is achieved by providing a command like

wait.until.t

where t is a number specifying the instant of simulated time at which the command is to be obeyed. If simulated time has already passed t, this is an error; if simulated time is equal to t, the command can be obeyed immediately; but if simulated time has not yet reached t, the command is delayed until it does. Let S be a process whose subprocesses contain instances of this command. Then we need to define a process SIMULATE (S), which describes the behaviour of S when executing in quasiparallel in simulated time.

Let p and q be (possibly compound) process names. Then clearly every trace s of SIMULATE(S) must satisfy

(1) If "p.waituntil.t1" occurs earlier in s than "p.waituntil.t2" then t1 must be not later than t2.

Also we wish to ensure that a "waituntil" command will not be obeyed while there is still something else to do at the current moment of simulated time i.e.

(2) If $s\sigma$ is in S, and σ is <u>not</u> a "waituntil" command, then s cannot be followed by a "waituntil" command in SIMULATE(S).

SIMULATE(S) can be defined as the subset of all traces in S which satisfy properties (1) and (2); a more constructive definition could also be given, but it is even more complicated, since it models the activity of an implementation.

Technical Note.

The SIMULATE function is not monotonic in the simple subset ordering of languages. It follows that it cannot be defined by any combination of the operators introduced previously in this paper.

Of course, realistic simulation will also require some facility for a process to find out the current value of simulated time; and some current capability of joining queue of processes waiting for service. Both of these facilities can be defined by extension of the model presented here.

Acknowledgements.

This work owes everything to the inspiration of Robin Milner, and the close collaboration of John R. Kennaway, Nissim Francez and Willem P. de Roever. Many defects have been removed from earlier drafts by the kind attention of Ann Yasuhara, Edsger W. Dijkstra and others.

References.

1 Dijkstra, E.W. Cooperating sequential processes, in Programming Languages ed. F. Genuys. Academic Press, New York, 1968 pp 43-112.

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

3 Francez, N. et al

4 Kaubisch, W.H. et al. Quasi parallel Programming, Software Practice and Experience. 6 (1976) pp 341 - 356.

5 Hoare, C.A.R. Communicating Sequential Processes. Comm ACM 21.8 (Aug. 1978) pp 666 - 677.

6 Hoare, C.A.R. Some Properties of Predicate Transformers. JACM 25.3 (July 1978) pp 461 - 480.

7 Milner, A.J.R.G.