# Communicating Sequential Processes.

C.A.R. Hoare

Department of Computer Science,
The Queen's University, Belfast.

MARCH 1977

Summary: This paper suggests that input and output are basic primitives of programming; and that parallel composition of communicating sequential processes is a fundamental program structuring method. When combined with a development of Dijkstra's guarded command, these concepts are surprisingly versatile. Their use is illustrated by sample solutions of a variety of familiar programming exercises.

Key words and phrases: programming, programming languages, programming primitives, program structures, parallel programming, concurrency, input, output, guarded commands, nondeterminacy, coroutines, procedures, multiple entries, multiple exits, classes, data representations, recursion, conditional critical regions, monitors, iterative arrays.

CR categories: 4.2, 4.22, 4.32.

This research was supported by a Senior Fellowship of the Science Research Council.

## 1. Introduction.

Among the primitive concepts of computer programming, and of the high level languages in which programs are expressed, the action of assignment is familiar and well understood. In fact, any change of the internal state of a machine executing a program can be modelled as an assignment of a new value to some variable part of that machine. However, the operations of input and output, which affect the external environment of a machine, are not nearly so well understood. Often they are added to a programming language only as an afterthought, designed without due regard to efficiency, security, or simplicity.

Among the structuring methods for computer programs, three basic constructs have received widespread recognition and use:- a repetitive construct (e.g., the while loop), an alternative construct (e.g., the conditional if.. then.. else), and normal sequential program composition (often denoted by semicolon). Less agreement has been reached about the design of other important program structures, and many suggestions have been made: subroutines (FORTRAN), procedures (ALGOL 60 [13] ), entries (PL/I), coroutines (UNIX [13] ), classes (SIMULA 67 [5] ), processes and monitors (Concurrent PASCAL [2] ), clusters (CLU [12] ), forms (ALPHARD [16] ), actors (Hewitt [1] ).

The traditional stored program digital computer has been designed primarily for deterministic execution of a single sequential program. Where the desire for greater speed has led to the introduction of parallelism, every attempt has been made to disguise this fact from the programmer, either by hardware itself (as in the multiple function units of the CDC 6600) or by the software (as in an I/O control package, or a multiprogrammed operating system). However, developments of processor technology suggest that a multiprocessor machine, constructed from a number of similar self-contained processors (each with its own store) may become appreciably more powerful, capacious, reliable, and economical than a machine which is disguised as a monoprocessor.

In order to use such a machine effectively on a single task, the component processors must be able to communicate and to synchronise with each other; and many methods of achieving this have been proposed. A widely adopted method of communication is by inspection and updating of a common store (as in ALGOL 68 [15]), PL/I, and many machine codes). However, this can create severe problems in the construction of correct programs; and it may lead to expense (e.g., crossbar switches) and unreliability (e.g., glitches) in a hardware implementation. A greater variety of methods has been proposed for synchronisation: semaphores [6], events (PL/I), conditional critical regions [10], monitors and queues (concurrent PASCAL [2] ), and path expressions [3]. Most of these are demonstrably adequate for their purpose, but there is no widely recognised criterion for choosing between them.

This paper makes an ambitious attempt to find a single simple solution to all these problems. The essential proposals are:

(1) Dijkstra's guarded commands [8] are adopted (with a slight change of notation) as sequential control structures, and as the sole means of introducing and controlling non-determinism.

(2) A parallel command, based on Dijkstra's <u>parbegin</u> [6], specifies concurrent execution of its constituent sequential commands (processes). All the processes start and end simultaneously. They may not communicate with each other by updating global variables.

(3) Simple forms of input and output commands are introduced. They are used for communication between concurrent processes.

(4) Such communication occurs when one process names another as destination for output <u>and</u> the second process names the first as source for input. In this case, the value to be output is copied from the first process to the second. There is <u>no</u> automatic buffering; in general, an input or output command is delayed until the other process is ready with the corresponding output or input. Such delay is invisible to the delayed program.

(5) Input commands are allowed to appear in guards. If they appear in several guards of a set of alternatives, the <u>first</u> such command which can be executed will determine the selection of the alternative, and the remaining guards will have no effect. Since only the selected guard has any actual effect, we avoid the problem of side effects of failed or unselected guards.

(6) A simple "pattern matching" mechanism is used to discriminate the structure of an input message, and to assign names to its components in a secure fashion.

The programs expressed in the proposed language are intended to be implementable both by a conventional machine with a single main store, and by a fixed network of processors connected by input/output channels (although very different optimisations are appropriate in the different cases). Consequently it is a rather static language: the text of a program determines a fixed upper bound on the number of processes operating concurrently; there is no recursion, and no facility for process-valued variables.

The concept of a communicating sequential process is shown in sections 3-6 to provide a method of expressing solutions to many simple programming exercises, which have been used before to illustrate the use of various other proposed programming language features. This suggests that this concept may constitute a synthesis of a number of familiar and new programming ideas.

However, this paper also ignores many serious problems. The most serious is that it fails to suggest any proof method to assist in the development and verification of correct programs. Secondly, it pays no attention to the serious problems of efficient implementation, particularly on a traditional sequential computer. It is probable that a solution to these problems will require:

(1) imposition of restrictions in the use of the proposed features,

(2) reintroduction of distinctive notations for the most common and useful special cases,

(3) the design of more appropriate hardware.

Thus the concepts and notations introduced in this paper (although described in the next section in the form of a programming language fragment) should not be regarded as suitable for use as a programming language, either for abstract or for concrete programming. They are at best only a partial solution of the problems tackled.

## 2. Concepts and notations.

The style of the following description is borrowed from ALGOL 60 [13]. Types, declarations, and expressions have not been treated; in the examples, a PASCAL-like notation [17] has been used. The curly braces { } have been introduced into BNF to denote none or more repetitions of the enclosed material.

```
<command> ::= <simple command>|<structured command>
<simple command> ::= <null command>|<assignment command>
                     |<input command>|<output command>
<structured command> ::= <alternative command>|<repetitive command>
                         |<parallel command>
<null command> ::= skip
<command list> ::= {<declaration>;|<command>;} <command>
```

A command specifies the behaviour of a device executing the command. Execution of a structured command involves execution of some or all of its constituent simple commands. Execution of a simple command may fail, and have no effect; or it may succeed, and have an effect on the internal state of the executing device (an assignment), or on its external environment (an output), or on both (an input). A null command has no effect and never fails.

Failure of a command appearing in a guard prevents selection of the command it guards (see 2.4). Any other failure prevents further execution of commands. Such failures are due to programming errors, detected by the executing device; their consequences will not be described here.

A command list specifies sequential execution of its constituent commands, in the order written. Each declaration introduces a fresh variable with a scope that extends from its place of occurrence to the end of the command list.

## 2.1 Assignment commands.

```
<assignment command> ::= <target variable> := <expression>
<expression> ::= <simple expression>|<structured expression>
<structured expression> ::= <constructor>(<expression list>)
<constructor> ::= <identifier>|<empty>
<expression list> ::= <empty>|<expression>{,<expression>}
<target variable> ::= <simple variable>|<structured target>
<structured target> ::= <constructor>(<target variable list>)
<target variable list> ::= <empty>|<target variable>{,<target variable>}
```

An assignment command specifies evaluation of its expression, and assignment of its value to the target variable. It fails if the expression is undefined, or if the structure of its value does not match the structure of the target variable.

A simple expression has a value which may be simple or structured. A structured expression has a structured value, with the specified constructor, and with a list of component values given by the constituent expressions of the expression list.

A simple target variable may be assigned any value of the same type. An assignment with a structured target specifies multiple assignments to its constituent simple variables of the corresponding components of the structured value of the expression. Consequently, the value of the target variable after a successful assignment is always the same as the value of the expression before the assignment. If this is not possible, the assignment fails.

A simple target variable matches any value of its type. A structured target variable matches a structured value, provided that:

(1) they have the same constructor

(2) the target variable list is the same length as the list of components of the value

(3) each target variable of the list matches the corresponding component of the value list.

Examples:

(1) x := x+1 - the value of x after the assignment is the same as the value of x+1 before.

(2) (x,y) := (y,x) - exchanges the values of x and y

(3) left := cons(left,right) - constructs a structured value and assigns it to left

(4) cons(left,right) := left - fails if left is an atom. Otherwise right gets the value of the second component of left, and left gets the value of its own first component

(5) insert(n) := insert(2*x+1) - equivalent to n := 2*x+1

(6) c := P( ) - assigns to c a "signal" with constructor P, and no components

(7) P( ) := c - fails if the value of c is not P( ); otherwise has no effect

(8) insert(n) := has(n) - fails, due to mismatch.

## 2.2 Input and output commands.

```
<input command> ::= <source>?<target variable>

<output command> ::= <destination>!<expression>

<source> ::= <process name>

<destination> ::= <process name>

<process name> ::= <identifier>|<identifier>(<subscripts>)

<subscripts> ::= <integer expression>{,<integer expression>}
```

Input and output commands specify communication between concurrently operating sequential processes. Such a process may be implemented in hardware as a special-purpose device (e.g. cardreader or lineprinter) or its behaviour may be specified by one of the constituent processes of a parallel command (see 2.3).

Communication occurs whenever:

(1) one process names another as the source for input
and (2) that other process names the first process as destination for output
and (3) the target variable of the input matches the value output.

On each such occasion, the two commands are executed simultaneously, and their combined effect is to assign the value of the expression in the output command to the target variable in the input command. A consequence of the requirement for synchronisation is that in general one of the two commands will have to wait for the other.

An input or output command fails if its source or destination is terminated. An output command fails if its expression is undefined; and an input command fails (with no effect) if its target variable does not match the value next output by the named destination to the given source. In this case, the output command is merely delayed, and does not fail.

Examples:

| | | |
|---|---|---|
| (1) cardreader?cardimage | – | from cardreader, read a card and assign its value (an array of characters) to the variable cardimage |
| (2) lineprinter!lineimage | – | to lineprinter, send the value of lineimage for printing. |

(3) X?(x,y) - from process named X, input two values and assign them to x and y

(4) DIV!(3*a+b,13) - to process DIV, output the two specified values.

Note: if a process named DIV issues command (3), and a process named X issues command (4), these are executed simultaneously, and have the same effect as the assignment:

```
(x,y) := (3*a+b,13)
```

(5) console(i)?x - from the ith element of an array of consoles, input a value and assign it to x

(6) console(j-1)!"A" - to the j-1th console, output character "A"

(7) X(i)?V() - from the ith of an array of processes X, input a signal V(); reject any other signal

(8) sem!P() - to sem, output a signal P()

## 2.3 Parallel commands.

```
<parallel command> ::= [<process>{||<process>}]
<process> ::= <process label> <command list>
<process label> ::= <empty>|<identifier> ::
                  |<identifier>(<constant subscript>{,<constant subscript>}) ::
<constant subscript> ::= <integer constant>|<range>
<integer constant> ::= <numeral>|<bound variable>
<bound variable> ::= <identifier>
<range> ::= <bound variable>:<integer constant>..<integer constant>
```

A parallel command specifies concurrent execution of its constituent processes. All of them are started simultaneously, and the parallel command terminates only when they are all terminated. Each process must be disjoint from every other process of the command. Two processes are disjoint if:

(1) neither of them assigns to any non-local variable mentioned in the other,
and (2) neither communicates with any non-local process with which the other also communicates (either directly or indirectly).

The intended effect of these restrictions is:

(1) Processes of a parallel command interact only by input and output commands.

(2) The non-local inputs and outputs of a parallel command are logically indistinguishable from those of an equivalent sequential command. Internal parallelism should be invisible from the outside.

Where a constant subscript is defined as a range, the effect is the same as writing out an array of processes, each with the same command list; and each with a different value of the bound variable, within the specified range. For example, X(i:1..n)::CL is equivalent to

$$X(1)::CL_1 \| X(2)::CL_2 \| \ldots \| X(n)::CL_n$$

where each $CL_j$ is formed from CL by replacing each occurrence of the bound variable i by the numeral j.

Consequently: (1) a bound variable must not appear (anywhere) as a target variable

(2) the rule of disjointness prohibits CL from updating non-local variables, or communicating with non-local processes. *

The process labels which prefix the processes of a parallel command are local to the command, and are used for communication between processes of that command. A process with empty name cannot engage in such communication. The process labels must be unique after the expansion described in the previous paragraph.

Examples:

(1) [cardreader?cardimage||lineprinter!lineimage]

Performs the input concurrently with the output, and terminates only when both operations are complete. The time taken may be as low as the longer of times taken by the constituent processes, i.e., the sum of its waiting and its transfer time.

(2) [west::DISASSEMBLE||X::SQUASH||east::ASSEMBLE]

The three processes may communicate with each other, using the names "west", "X", and "east" as sources and destinations for input and output. The capitalised words stand for command lists which will be defined in later examples.

(3) [room::ROOM||fork(i:0..4)::FORK||phil(i:0..4)::PHIL]

There are eleven processes. The behaviour of "room" is specified by the command list ROOM. The behaviour of the five processes fork(0), fork(1), fork(2), fork(3), fork(4), is specified by the command list FORK, within which the bound variable i may be used to indicate the identity of the particular fork. Similar remarks apply to the five processes PHIL.

---

* It may be desirable to relax this rule to permit each process to access and update an array element provided that this element is subscripted by the full process index. Thus it is guaranteed that each process is actually operating on a different element of the array.

## 2.4 Alternative and repetitive commands.

```
<repetitive command> ::= *<alternative command>

<alternative command> ::= [<guarded command>{□<guarded command>}]

<guarded command> ::= <guard>→<command list>

                    |(<range>)<guard>→<command list>

<guard> ::= <guard list>|<guard list>,<input command>

                        |<input command>

<guard list> ::= <guard element>{;<guard element>}

<guard element> ::= <boolean expression>|<declaration>
```

A guard is executed from left to right. A Boolean expression is evaluated; if the value is false, execution fails; otherwise it has no effect. A declaration introduces variables with a scope extending to the end of the guarded command. An input command is executed only if it does not fail. Thus, a guard which fails is discontinued, and has no effect.

A guarded command with a range takes the form (i:m..n)G→CL. It is exactly equivalent to the n-m+1 guarded commands:

$$G_m \rightarrow CL_m \Box G_{m+1} \rightarrow CL_{m+1} \Box \ldots \Box G_n \rightarrow C_n$$

where $G_j \rightarrow CL_j$ is the result of replacing every occurrence of the bound variable i in G→CL by the numeral j.

An alternative command specifies concurrent execution of all its guards. If all the guards fail, the alternative command fails. Otherwise, the guarded command with the earliest successfully completed guard is selected and executed, followed by execution of its command list; execution of all the other guards is discontinued, without any effect. Since the relative speeds of execution of the guards is not determined, neither is the choice between two or more non-failing guards.

A repetitive command specifies as many iterations as possible of its constituent alternative command. Thus, if all of its guards fail, the repetitive command terminates. Otherwise, the alternative command is executed once, and this is followed by execution of the whole repetitive command again.

Note that when a guard ends with an input command, execution of the guard may be delayed waiting for the matching output. An alternative command is delayed if all its guards are delayed; however, if all the awaited outputs are ready and fail to match, the alternative command fails.

A repetitive command is also delayed if its guards are all delayed; and it terminates only when all awaited input guards have failed (as a result of mismatch or termination).

Examples:

```
(1)  [x≥y→m:=x□y≥x→m:=y]
```

If x≥y, assign x to m; if y≥x assign y to m; if both x≥y and y≥x, either assignment can be executed.

```
(2)  i:=0;*[i<size;content(i)≠n→i:=i+1]
```

The repetitive command scans the elements content(i), for i=0,1,.., until either i≥size, or a value equal to n is found.

```
(3)  *[c:character;west?c→east!c]
```

This reads all the characters output by west, and outputs them one by one to east. The repetition terminates when the process west terminates, or offers a value which is not a character.

```
(4)  *[(i:1..10)console(i)?c→X!(i,c);console(i)!ack()]
```

Repeatedly inputs to c any of ten consoles. The identity of the console and the content of its message are then output to X, and an acknowledgement is sent back to the originating console.

```
(5)  *[n:integer;X?insert(n)→INSERT
      □n:integer;X?has(n)→SEARCH;X!(i≥size)
      ]
```

On each iteration, accept from X <u>either</u> a request to "insert(n)", (followed by INSERT), <u>or</u> a request "has(n)", (followed by SEARCH, and an output of an answer back to X). Of course, the choice between these alternatives will be determined by process X.

The repetitive command terminates when X terminates (or before, if X offers non-matching output).

```
(6)  *[X?V()→val:=val+1
      □val>0;Y?P()→val:=val-1
      ]
```

On each iteration, accept <u>either</u> a V() signal from X and increment val, <u>or</u> a P() signal from Y, and decrement val. But the second alternative cannot be selected unless val is positive (after which val will remain invariantly nonnegative). When val>0, the choice depends on the relative speeds of X and Y, and is not determined. The repetitive command will terminate when both X and Y are terminated (or before).

## 3. Coroutines.

In parallel programming, coroutines appear as a more fundamental program structure than subroutines, which can be regarded as a special case (treated in the next section).

### 3.1 COPY

Problem: write a process X to copy characters output by process west to process east.

Solution:
```
X::*[c:character;west?c→east!c]
```
Notes:

(1) When west terminates, the input "west?c" will fail, causing termination of the repetitive command, and of process X. Any subsequent input command from east will fail.

(2) Process X acts as a single-character buffer between west and east. It permits west to work on production of the next character, before east is ready to input the previous one.

### 3.2 SQUASH

Problem: adapt the previous program to replace every pair of consecutive asterisks "**" by an upward arrow "↑". You may assume that the last character input is not an asterisk.

Solution:
```
X::*[c:character;west?c→
          [c≠asterisk→east!c
          □c=asterisk→west?c;
                    [c≠asterisk→east!asterisk;east!c
                    □c=asterisk→east!upward arrow
     ] ]            ]
```
Notes:

(1) Since west does not end with asterisk, the second "west?c" will not fail.

(2) As an exercise, adapt this process to deal sensibly with input which ends with an odd number of asterisks.

## 3.3 DISASSEMBLE

Problem: to read cards from a cardreader and output to process X the stream of characters they contain. An extra space should be inserted at the end of each card.

Solution:
```
*[cardimage:(1..80)character ;
         cardreader?cardimage→
         i:integer; i:=1;
         *[i≤80→X!cardimage(i):=i+1];
         X!space
 ]
```

Note:
(1) "(1..80)character" declares an array of 80 characters, with subscripts ranging between 1 and 80.

## 3.4 ASSEMBLE

Problem: read a stream of characters from a process X and print them in lines of 125 characters on a lineprinter. The last line should be completed with spaces if necessary.

Solution:
```
lineimage:(1..125)character;
i:integer; i:=1;
*[c:character;X?c→
            lineimage(i):=c;
            [i≤124→i:=i+1
            □i=125→lineprinter!lineimage; i:=1
 ]          ];
 [i=1→skip
 □i>1→*[i≤125→lineimage(i):=space; i:=i+1];
      lineprinter!lineimage
 ]
```

Note:
(1) When X terminates, so will the first repetitive command of this process. The last line will then be printed, if it has any characters (and sometimes if it does not).

## 3.5. Reformat.

Problem: read a sequence of cards of 80 characters each, and print the characters on a lineprinter at 125 characters per line. Every card should be followed by an extra space, and the last line should be completed with spaces if necessary.

Solution: [west::DISASSEMBLE||X::COPY||east::ASSEMBLE]

Notes:

(1) The text of the processes is found in previous sections.

(2) This elementary problem is difficult to solve elegantly without coroutines.

## 3.6. Conway's Problem. [4].

Problem: adapt the above program to replace every pair of consecutive asterisks by an upward arrow.

Solution: [west::DISASSEMBLE||X::SQUASH||east::ASSEMBLE]

## 4. Subroutines and data representations.

A conventional subroutine can be readily implemented as a coroutine, provided that:

(1) its parameters are called "by value" and "by result"

(2) it updates no nonlocal variables used in its calling program.

Like a FORTRAN subroutine, a coroutine may retain the values of local variables (<u>own</u> variables, in ALGOL terms); and it may use input commands to achieve the effect of "multiple entry points", in a safer way than PL/I. Thus a coroutine can be used like a SIMULA class instance as a concrete representation for abstract data.

A coroutine acting as a subroutine is a process operating concurrently with its user process in a parallel command:

```
[subr::SUBROUTINE||X::USER]
```

The SUBROUTINE will contain (or consist of) a repetitive command:

```
*[X?{value params};...;X!{result params}]
```

where ... computes the results from the values input.
The USER will call the subroutine by a pair of commands:

```
subr!{arguments};...;subr?{results}
```

Any commands between these two will be executed concurrently with the subroutine.

A multiple-entry subroutine, acting as a representation for data, [A4] will represent each entry by an alternative input to a structured target, with the entry name as constructor; e.g.

```
*[X?entry1(values)→...
 □X?entry2(values)→...
 ]
```

The calling process X will determine which of the alternatives is activated on each repetition. When X terminates, so does this repetitive command. A similar technique in the user program can achieve the effect of multiple exits.

A recursive subroutine can be represented by an array of processes, one for each level of recursion. The calling process is level zero; each activation communicates its parameters and results with its predecessor, and calls its successor if necessary:

```
[recsub(0)::USER||recsub(i:1..reclimit)::RECSUB]
```

The user will call the first element of recsub:

```
recsub(1)!{arguments}...;recsub(1)?{results};
```

The imposition of a fixed upper bound on recursion depth is necessitated by the "static" design of the language.

In this section, we assume each subroutine is used only by a single user process. The next section shows how this restriction can be lifted.

## 4.1 Function: division with remainder.

Problem: construct a process to represent a function-type subroutine, which accepts two positive integer parameters, x and y, and returns their integer quotient and remainder. Efficiency is of no concern.

Solution:
```
[DIV::*[x,y:integer;X?(x,y)→
            quot,rem:integer; quot:=0; rem:=x;
           *[rem≥y→rem:=rem-y; quot:=quot+1];
            X!(quot,rem)
         ]
||X:: USER PROG
]
```

## 4.2 Recursion: factorial.

Problem: compute a factorial by the recursive method, to a given limit.

Solution:
```
[fac(i:1..limit)::
 *[n:integer;fac(i-1)?n→
   [n=0→fac(i-1)!1
   □n>0→fac(i+1)!n-1;
      r:integer;fac(i+1)?r;fac(i-1)!(n*r)
  ] ]
||fac(0)::USER PROG
]
```

## 4.3 Data representation: small set of integers.

Problem: to represent a set of not more than 100 integers as a process S, which accepts two kinds of instruction from its calling process:

(1) S!insert(n) - insert the integer n in the set

(2) S!has(n);...;S?b - b is set true if n is in the set, and false otherwise.

The initial value of the set is empty.

Solution: S::

```
content:(0..99)integer; size:integer; size:=0;
*[n:integer; X?has(n)→SEARCH; X!i≥size
 □n:integer; X?insert(n)→SEARCH;
            [i<size→skip
            □i=size,size<100→
                     content(size):=n; size:=size+1
 ]          ]
```

where SEARCH is an abbreviation for:

```
i:integer; i:=0;
*[i<size; content(i)≠n→i:=i+1]
```

Notes:

(1) The alternative command with guard "size<100" will fail if an attempt is made to insert more than 100 elements.

(2) The activity of insertion will in general take place concurrently with the calling process. However, any subsequent instruction to S will be delayed until S is ready to accept it.

## 4.4. Scanning s set.

Problem: extend the solution to 4.3, by providing a fast method for scanning all members of the set, without changing the value of the set. The user program will contain repetitive commands of the form:

```
         S!scan( );
        *[x:integer, S?next(x)→  ... deal with x ....];
         S?noneleft( )
```

where the first line sets the representation into a scanning mode; the second line serves as a for statement inputting the successive members of x from the set, and inspecting them; until finally the representation sends a signal that there are none left.

The body of the repetitive command is not permitted to communicate with S in any way.

Solution: add a third guarded command to the outer repetive command of S :

```
...□X?scan( )→  i:integer; i:=0;
      *[i<size→X!next(content(i)); i:=i+1];

       X!noneleft( )
```

Note:

(1) As an exercise, extend the above solution to permit the user program to test membership of the set S while still inside the body of the loop. This will require a further repetitive command inserted just after i:=i+1.

## 4.5. Recursive data representation: small set of integers.

Problem: same as above; but an array of processes is to be used to achieve high parallelism. The $i^{th}$ process should contain the $i^{th}$ largest member; when it does not contain any number, it should answer "false" to all enquiries about membership. On the first insertion, it changes to a second phase of behaviour, in which it deals with instructions from its predecessor, passing some of them on to its successor. The calling process will be named S(O).

```
Solution:  S(i:1..100)::

           *[n:integer; S(i-1)?has(n)→S( 0  )!false
            □n:integer; S(i-1)?insert(n)→
                 *[m:integer S(i-1)?has(m)→
                      [m≤n→S(0)!(m=n)
                      □m>n→S(i+1)!has(m)
                      ]
                  □m:integer; S(i-1)?insert(m)→
                      [m<n→S(i+1)!insert(n); n:=m
                      □m=n→skip
                      □m>n→S(i+1)!insert(m)
            ]    ]    ]
```

Notes:

(1) The calling process S(0) enquires whether n is a member by the commands

S(1)!has(n); [(i:1..100)S(i)?b→skip]

The appropriate process will respond to the input command by the output command in line 2 or line 5. This trick avoids passing the answer back "up the chain".

(2) Many insertion operations can proceed in parallel; yet any subsequent "has" operation will be performed correctly.

(3) Exercise: extend the above solution to respond to a command to yield the least member of the set, and then to remove it from the set. The user program will invoke the facility by a pair of commands:

```
        S[0]!least( ); [x:integer; S[0]?x→  ... x ...
                        □S 0 ?noneleft( )→  ...
                        ]
```

or in a "for loop" by:

```
    S[0]!least( ); *[x:integer; S[0]?x→  ... x ...  ; S[0]?noneleft( );
```

Hint: introduce a boolean variable b, initialised to true; and prefix this to all the guards of the inner loop. After responding to a !least( ) command from its predecessor, each process returns its contained value n, asks its successor for its least, and stores the response in n. But if the successor returns "noneleft", b is set false, and the inner loop terminates. The process therefore returns to its initial state. (Solution due to David Gries).

## 5. Monitors and scheduling.

This section removes the restriction imposed in the previous section that the user of a subroutine must be a single process. In fact, a monitor can be regarded simply as a single process which communicates with more than one user process. Of course, each user process must have a different name (e.g. producer, consumer) or a different subscript (e.g. X(i)), and each communication with a user must identify its source or destination uniquely.

Consequently, when a monitor is prepared to communicate with any of its user processes (i.e. whichever of them calls first), it will use a guarded command with a range, e.g., :

```
*[(i:1..100) X (i)?value parameters →...; X(i)!results]
```

If the monitor is not prepared to accept input from some particular user (e.g. X(j)) on a given occasion, the input command may be preceded by a boolean guard e.g.

```
* [(i:1..100) i ≠ j, X(i)? values → ...; j:=i]
```

Any attempted output from X(j) will be delayed until a subsequent iteration, after the output of some other process X(i) has been processed.

Similarly, conditions can be used to delay acceptance of inputs which would violate scheduling constraints, postponing them until some later occasion when some other process has brought the monitor into a state in which the input can validly be accepted. This technique is similar to a conditional critical region (Hoare ); and it obviates the need for special synchronising variables such as events, queues, or conditions.

### 5.1. Bounded buffer.

Problem: construct a buffering process X to smooth variations in the speed of output of a producer process and input by a consumer process. The consumer contains pairs of commands "X!more( ); X?p", and the producer contains commands of the form X!p. The buffer should contain up to ten portions.

Solution: X::

```
    buffer: (0..9) portion;
    in, out:integer; in:=0; out :=0;
    comment 0 ≤ out ≤ in ≤ out + 10;
    * [in < out+10, producer ? buffer (in mod 10) → in := in+1
      out < in, consumer ? more ( ) → consumer ! buffer (out mod 10) ;
                                        out := out+1
      ]
```

Notes:

(1) When out ≤ in < out+10, the selection of the alternative in the repetitive command will depend on whether the producer produces before the consumer consumes, or vice-versa.

(2) When out = in, the second alternative cannot be selected, even if the consumer is ready with its command "X!more( )". However, after the producer has produced its next portion, the consumer's request can be granted on the next following repetition.

(3) Similar remarks apply to the producer, when in = out+10.

5.2. Integer semaphore.

Problem: to implement an integer semaphore S, shared among an array X(i:1..100) of client processes. Each process may increment the semaphore by S!V( ), or decrement it by S!P( ); but the latter command must be delayed if the value of the semaphore is not positive.

Solution:

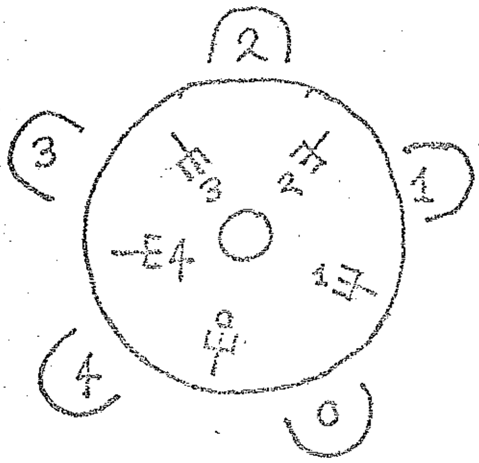
```
S::val:integer ; val := 0;
    * [(i:1..100) X (i) ? V ( ) → val := val+1
       (i:1..100) val > 0, X(i) ? P( ) → val := val-1
      ]
```

Note:

(1) In this process, no use is made of knowledge of the subscript i of the calling process.

## 5.3. Dining philosophers. (Problem due to E.W. Dijkstra).

Problem: Five philosophers spend their lives thinking and eating. The philosophers share a common dining room where there is a circular table surrounded by five chairs, each belonging to one philosopher. In the centre of the table there is a large bowl of spaghetti, and the table is laid with five forks:



On feeling hungry, a philosopher enters the dining room, sits in his own chair, and picks up the fork on the left of his place. Unfortunately, the spaghetti is so tangled that he needs to pick up and use the fork on his right as well. When he has finished, he puts down both forks, and leaves the room. The room should keep a count of the number of philosophers in it.

Solution: The behaviour of the ith philosopher may be described as follows:

```
PHIL = *[true→
         THINK;
         room!enter( );
         fork(i)!pickup( ); fork((i+1) mod 5)!pickup( );
         EAT;
         fork(i)!putdown( ); fork((i+1) mod 5)!putdown( );
         room!exit( )
        ]
```

The fate of the ith fork is to be alternately picked up and put down by a philosopher sitting on either side of it

```
FORK =
     *[phil(i)?pickup( )→phil(i)?putdown( )
      □phil((i-1) mod 5)?pickup( )→phil((i-1) mod 5)?putdown( )
      ]
```

The story of the room may be simply told:

```
ROOM = occupancy:integer; occupancy:=0;
       *[(i:0..4)phil(i)?enter( )→occupancy:=occupancy+1
        □(i:0..4)phil(i)?exit( )→occupancy:=occupancy-1
        ]
```

All these components operate in parallel:

```
[room::ROOM||fork(i:0..4)::FORK||phil(i:0..4)::PHIL]
```

(1) The solution given above does not prevent all five philosophers from entering the room, each picking up his left fork, and starving to death, because he cannot pick up his right fork.

(2) Exercise: adapt the above program to avert this sad possibility. This can be done by preventing more than four philosophers from entering the room. (Solution due to E.W. Dijkstra).

## 6. Miscellaneous.

This section contains further examples of the use of communicating sequential processes for the solution of some less familiar problems; a parallel version of the sieve of Eratosthenes, and the design of an iterative array. The proposed solutions are even more speculative than those of the previous sections, and they have not been tested by a practical implementation. In each case, the solutions have been constructed not for efficiency or realism, but for simplicity and perhaps elegance.

### 6.1. Prime Numbers: the sieve of Eratosthenes.

Problem: to print in ascending order all primes less than 100000. Use an array of processes, SIEVE, in which each process inputs a prime from its predecessor, and prints it. The process then inputs an ascending stream of numbers from its predecessor and passes them on to its successor, suppressing any that aremultiples of the original prime.

Solution:

```
[SIEVE (i:1..100)::
  p,mp:integer;
  *[SIEVE (i-1)?p→
      print!p;
      mp:=p; comment mp is a multiple of p;
      *[m:integer; SIEVE(i-1)?m→
            *[m>mp mp:=mp+p];
            [m=mp→skip
           []m<mp→SIEVE(i+1)!m

    ]   ]   ]

        ||SIEVE(0)::print!2; n:integer; n:=3;
                  * [n<100000 → SIEVE(1)!n; n:=n+2;]

        ||SIEVE(101)::   * [n:integer; SIEVE(100)?n → print!n]
        || print:: * [(i:0..101) n:integer; SIEVE(i)?n → ...   ]
        ]
```

Note: (1) This beautiful solution was contributed by David Gries.

(2) It is algorithmically similar to the program developed in [7, pp.27-38].

6.2 An iterative array: matrix multiplication.

Problem: A square matrix A of order 3 is given.
Three streams are to be input, each stream representing a column of an array IN. Three streams are to be output, each representing a column of the product matrix IN×A. After an initial delay, the results are to be produced at the same rate as the input is consumed. Consequently, a high degree of parallelism is required.

Solution: The solution takes the form shown in fig.1. Each non-border node inputs a vector component from the west and a partial sum from the north. It outputs the vector component to its east, and an updated partial sum to the south. The input data is produced by the west border nodes, and the desired results are consumed by south border nodes. The north border is a constant source of zeroes; and the east border is just a sink.

No provision is made in this program for termination, nor for changing the values of the array A.

There are twenty one nodes, in five groups, comprising the central square and the four borders:

```
 [M(i:1..3,0):: west
||M(0,j:1..3):: north
||M(i:1..3,4):: east
||M(4,j:1..3):: south
||M(i:1..3,j:1..3):: centre
 ]
```

The west and south borders are processes of the user program; the remaining processes are:

```
north = *[true M(1,j)!0]
east  = *[x:real; M(i,3)?x→skip]
centre= *[x:real; M(i,j-1)?x→
                M(i,j+1)!x; sum:real;
                M(i-1,j)?sum;   M(i+1,j)!A(i,j)*x+sum
       ]
```

$$\begin{tikzcd}
 & \boxed{(0,1)} \arrow[d, "0"] & \boxed{(0,2)} \arrow[d, "0"] & \boxed{(0,3)} \arrow[d, "0"] & \\
\boxed{(1,0)} \arrow[r, "x"] & \boxed{(1,1)} \arrow[r, "x"] \arrow[d, "A_{11}x"] & \boxed{(1,2)} \arrow[r, "x"] \arrow[d, "A_{12}x"] & \boxed{(1,3)} \arrow[r, "x"] \arrow[d, "A_{13}x"] & \boxed{(1,4)} \\
\boxed{(2,0)} \arrow[r, "y"] & \boxed{(2,1)} \arrow[r, "y"] \arrow[d, "A_{11}x+A_{21}y"] & \boxed{(2,2)} \arrow[r, "y"] \arrow[d, "A_{12}x+A_{22}y"] & \boxed{(2,3)} \arrow[r, "y"] \arrow[d, "A_{13}x+A_{23}y"] & \boxed{(2,4)} \\
\boxed{(3,0)} \arrow[r, "z"] & \boxed{(3,1)} \arrow[r, "z"] \arrow[d, "A_{11}x+A_{21}y+A_{31}z"] & \boxed{(3,2)} \arrow[r, "z"] \arrow[d, "A_{12}x+A_{22}y+A_{32}z"] & \boxed{(3,3)} \arrow[r, "z"] \arrow[d, "A_{13}x+A_{23}y+A_{33}z"] & \boxed{(3,4)} \\
 & \boxed{(4,1)} & \boxed{(4,2)} & \boxed{(4,3)} &
\end{tikzcd}$$

$$\begin{tikzcd}
 & N & \\
W \arrow[r, dash] & \cdot \arrow[u] \arrow[r, dash] \arrow[d, dash] & E \\
 & S &
\end{tikzcd}$$

Fig 1.

## Conclusion.

This paper has suggested that input, output and concurrency should be regarded as primitives of procedural programming, which underlie many familiar and less familiar programming concepts. However, it would be unjustified to conclude that these primitives can replace the other concepts in a programming language. Where a more elaborate construction (such as a procedure or a monitor) has properties which are more simply provable, and can also be implemented more efficiently, there is a strong case for including in a programming language a special notation for that construction. The fact that the construction can be defined in terms of simpler underlying primitives is a useful guarantee that its inclusion is logically consistent with the remainder of the language.

It is tempting to explore further extensions to the ideas presented in this paper. For example, it may be possible to define arrays of processes with no *a priori* bounds on their number. Another possibility would be to allow other commands besides input commands to appear in guards, for example, assignment commands or even output commands. However, I fear that such extensions may impede the goals of provability and efficient implementation; and I suspect that the more promising line for future research would be to restrict the generality of the primitives, rather than to extend them.

## Acknowledgements.

The research reported in this paper has been encouraged and supported by a Senior Fellowship of the Science Research Council of Great Britain.

The technical inspiration was due to Edsger W. Dijkstra[9], and the paper has been improved in presentation and content by valuable and painstaking advice from D. Gries, D.Q.M. Fay, Edsger W. Dijkstra, N. Wirth, and Robert Milne.

The role of IFIP W.G.2.3 as a forum for presentation and discussion is acknowledged with pleasure and gratitude.

REFERENCES:

[1] Atkinson, R. and Hewitt, C.
Synchronisation in actor systems.
MIT room 813, Working Paper 83, Nov. 1976.

[2] Brinch Hansen, P.
The programming language Concurrent Pascal.
IEEE Trans. Soft. Eng. 1,2 (June 1975), 199-207.

[3] Campbell, R.H., Habermann, A.N.
The specification of process synchronisation by path expressions.
Lecture Notes in Computer Science, Springer 1974, 89-102.

[4] Conway, M.E.
Design of a separable transition-diagram compiler.
Comm. ACM 6,7. ( ) 396-408.

[5] Dahl, O-J. et al. SIMULA 67, common base language.
Norwegian Computing Centre, Forskningveien, Oslo (1967).

[6] Dijkstra, E.W.
Co-operating Sequential Processes, in
Programming Languages (ed. F. Genuys). Academic Press (1968).

[7] Dijkstra, E.W.
Notes on Structured Programming, in
Structured Programming, Academic Press (1972), 1-82.

[8] Dijkstra, E.W.
Guarded commands, nondeterminacy, and formal derivation of programs.
Comm.ACM, 18, 8 (Aug.1975). 453-457.

[9] Dijkstra,E.W.
Verbal communication. Marktoberdorf, Aug. 1975.

[10] Hoare, C.A.R.
Towards a theory of parallel programming, in
Operating Systems Techniques, Academic Press 1972, 61-71.

[11] Hoare, C.A.R.
Proof of correctness of data representations.
Acta Informatica 1, 271-281 (1972).

[12] Liskov, B.H.
A note on CLU.
MAC TR Massachusetts Inst. of Technology, June 1975.

[13] Naur, P. (ed.). Report on the algorithmic language ALGOL 60.
Comm.ACM 3 (May 1960), 299-314.

[14] Thompson,K.
The UNIX command language, in
Structured Programming, Infotech 1976, 375-384.

[15] van Wijngaarden (ed.)
Report on the Algorithmic Language ALGOL 68.
Numerische Mathematik 14 1969, 79-218.

[16] Wulf, W.A., London, R.L., Shaw, M.
Abstraction and verification in ALPHARD.
Dept.of Computer Science, Carnegie-Mellon University (June 1976).

[17] Wirth, N.
The Programming Language PASCAL.
Acta Informatica 1,1.