Communicating Sequential Processes.

C.A.R. Hoare,

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

DRAFT : August 1976. (Replaces:
An investigation into the structure of computations,
April 1976)

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, classes, data representations, recursion, conditional critical regions, monitors.

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; and 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, there are three basic constructs which have received widespread recognition and use; namely, a repetitive construct (e.g., the while loop), an alternative construct (e.g., the conditional if .. then .. else), and normal sequential program composition (usually denoted by semicolon). However, less agreement has been reached about the design of other important program structures, and many suggestions have been made: subroutines (FORTRAN), procedures (ALGOL 60), entries (PL/I), coroutines (UNIX), classes (SIMULA 67), processes and monitors (Concurrent PASCAL), clusters (CLU), modules (ALPHARD). None of these are wholly satisfactory in themselves; and yet their combination would be even more problematic.

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 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 of a multiprogramming operating system. However, developments of processor technology suggest that a multiprocessor machine, constructed from a number of similar selfcontained processors, 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, it is necessary that the component processors should 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, PL/I, and many machine codes); however, this can create severe problems in the construction of correct programs; and it may lead to expense and unreliability (e.g., glitches) in a hardware implementation. A greater variety of methods has been proposed for synchronisation: semaphores (Dijkstra), events (PL/I), conditional critical regions (Hoare), monitors and queues (concurrent PASCAL), condition variables (Hoare), and path expressions (Haberman). 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 tackle these problems, and to find a single simple solution to them all. Input and output are regarded as programming primitives like assignment; indeed output may be likened to a half-assignment (without a left hand side), and input to the other half. Under appropriate conditions, an input from one process is synchronised with an output from another; and their combined effect is the same as the corresponding whole assignment. Input and output provide the sole method of communication between processes running in parallel; they are also the main method of synchronisation. Additional synchronisation is achieved by the use of guards in a guarded command (Dijkstra).

The concept of a communicating sequential process appears to provide a neat method of expressing the 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 communicating sequential processes may constitute a simple 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 the (1) imposition of severe restrictions in the use of the proposed features, (2) reintroduction of distinctive notations for common and useful special cases, and even (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.

This section introduces the concepts of the paper, and suggests a notation for expressing them. The style of description is borrowed from (ALGOL 60), but it is quite incomplete. In particular, no proper treatment is given of expressions, types, or declarations. In the example problems, a notation similar to that of PASCAL (Wirth) has been adopted.

```
        <command>  ::=   <simple command>    |  <structured command>
<simple command> ::=   <skip command>  |
                       <assignment command>  |  <input command> | <output command>
<structured command> ::=  <alternative command> | <repetitive command>|<parallel command>
                          <recursive command> |<recursive call>

<skip command> ::=  skip
```

A command is a specification of a computer process, to be executed by a machine. Execution of a structured command involves execution of some or all of its constituent simple commands. Execution of a command may fail; or it may have an effect on the internal state of the machine (e.g., assignment), or on its environment (e.g., output), or on both (e.g., input).

A skip command has no effect and never fails.

2.1. Assignment commands.

Syntax.

```
<assignment command> ::=  <target variable> :=<expression>

  <expression>::=  <simple expression>  | <structured expression>

  <structured expression>::=   <constructor>     (< expression list> )

  <expression list>::=  <empty> | <expression>,   <expression list>

  <constructor> ::=   <identifier>      |     <empty>

  <target variable>::=  <simple variable>   |    <structured variable>

  <structured variable> ::=   <constructor>   (<target variable list>)

  <target variable list> ::=      <empty>   |   <target variable>,
                                                 <target variable list>
```

Examples

```
 x:=x+1                              insert(n):=insert(2×x+1)
(x,y):=(y,x)                         P( ):=c
left:=cons(left,right)               c:=P( )
cons(left,right):=list 1
cons(x,cons(y,z)):=list 2
```

An assignment command specifies evaluation of its expression, and assignment to the target variable of the value it denotes. The command fails if the expression fails to denote a value, or if the structure of the value does not match the structure of the target variable.

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

A simple target variable matches any value of the same type. A structured target variable matches a structured value provided that (1) they have the same constructor; (2) the target variable list has the same length as the list of components of the value; and (3) each target variable of the list matches the corresponding component of the value list. Given this match, the effect of the assignment is to assign to each target variable the corresponding component value; in case of mismatch, the assignment fails.

2,2. Input and output commands.

Syntax.

```
<input command> ::=  <source> ?  <target variable>
<output command>::=  <destination> ! <expression>
<source>::  <process name>
<destination>::=  <process name>
<process name>::=   <process identifier> |
                    <process array identifier>     <subscript>

<subscript>::=    (<integer   expression>)
```

Examples:	cardreader? cardimage	lineprinter!lineimage
	west?c	X!m × n
	X? (x,y)	X! (rem,quot)
	X? insert(n)	s!insert(2×x+1)
	producer(i)?x	consumer(j)!buffer(out)
	X(i)?V( )	sem!P( )

An input command is delayed if necessary until the named source is ready to transmit a value; if the value matches the target variable, it is accepted and assigned to that variable. An input command fails in the case of mismatch, or if the named source is terminated.

An output command is delayed if necessary until the named destination is ready to accept a value. The expression is then evaluated, and its value is transmitted. An output command fails if the value does not match the target variable of the input, or if the expression fails in evaluation.

A process name either names a device external to the machine executing the command (e.g., cardreader, lineprinter), or it names a process to which that name is prefixed. The subscript following a process array identifier selects a particular element of that process array. Its value must be less than the number of processes in the array.

2.3. Parallel Commands.

<process> ::= <single process> | <array of processes>

<single process> ::= <process identifier>: <command>

<array of processes> ::= <process array identifier>: <process array>

$$\langle\text{process array}\rangle ::= \prod_{\langle\text{bound variable}\rangle}^{\langle\text{limit}\rangle} \langle\text{command}\rangle$$

<limit> ::= <integer constant> | < bound variable> | <integer constant expression>

<bound variable> ::= <identifier >

<process set> ::= <process> || <process> | <process set> || <process>

<parallel command> ::= [<process set>]

Examples [fac:F || X: [fac! n-1; m:integer; fac?m; X!m x n]]

[room:ROOM || fork: $\prod_{i}^{4}$ FORK || phil: $\prod_{i}^{4}$ PHIL]

A parallel command specifies parallel (concurrent) execution of its constituent processes. It is defined only if these processes are <u>disjoint</u>, in the sense that none of them contains as a target variable any variable mentioned in any other process of the command. The constituent command of a process array may contain as target variable only variables declared within that command. Consequently, each element of the array is disjoint from every other element.

A process array takes the form:

$$X: \prod_{i}^{n} C$$

It denotes an array of n processes; one for each value of i between 0 and n-1. It may be regarded as equivalent to the parallel set

$$X(0):C_0^i \;||\; X(1): C_1^i \;||\ldots||\; X(N-1): C_{n-1}^i$$

where C_j^i is the result of replacing in C every occurrence of the bound variable i by the numeral j. The bound variable i is local to C, and it must not appear as a target variable.

Each process of a parallel command is prefixed by its name. The scope of this name is confined to the other processes of the parallel command. Consequently,

(1) No process can communicate with itself

(2) Any occurrence of a process name within the process it names is either local to the named process, or global to the entire parallel command.

Communication between processes of a parallel command occurs whenever one process names another process as the source for an input command, and that other process names the first as the destination of an output command. On each such occasion, the two commands are executed simultaneously, and their combined effect is that of an assignment of the expression specified in the output command to the target variable specified in the input command.

2.4. Guarded commands.

Syntax.

```
<guarded command>::=    <guard>    →    <command list>
<guard>::=  <guard element>  | <guard element>; <guard list> |<declaration>; <guard list
<command list>::=    <command> |<command>;  <command list>|<declaration>; <command list>
<guard element>::=  < boolean expression> |   <assignment command>    |
                    < input command >
```

Examples.

```
                 x ≥ y  →   m:=x
                 x > y  → (x,y):= (y,x)

          c: character  ; west?c → n:= n+1;   east ! c

                 i ≥ size;  size < N → content(size):=n; size:=size + 1
                 out < m; consumer?more( ) → consumer!buffer(out);out:=out+1
                 x:integer; client(i)?x;  val+x ≥ 0 → val:=val + x
```

A guarded command specifies sequential execution of the guard elements of its guard, followed by sequential execution of the commands of the command list. A boolean expression in a guard is evaluated; if the result is <u>true</u>, it has no effect, and otherwise it fails.

A declaration introduces a variable to be used subsequently in the same guarded command. Its scope extends from its place of occurrence to the end of the guarded command in which it occurs. Every variable which appears as a target variable in a guard must have been declared previously in the same guard. Consequently, no guard can change any non-local variable. Furthermore, no guard may contain more than one input from the same source.

A guard is said to be <u>ready</u> if its execution would not be delayed. A guard is said to be feasible if it is ready and its execution would not fail. A guarded command is executed only if its guard is feasible, and only when it is ready. Consequently, a guard is never <u>partly</u> executed; it is either executed as a whole, or not at all.

Declarations followed immediately by input to the declared variable may be abbreviated as follows:

west?c:character	for	c:character; west?c
X?(x,y:integer)	for	x,y:integer; X?(x,y)
X?has(n:basetype)	for	n:basetype;X?has(n)

2.5. Alternative and repetitive commands.

Syntax.

<guarded command set> ::= <guarded element> | <guarded command set> $\square$ <guarded element>

<guarded element> ::= <guarded command> $\square$ <guarded array>

<guarded array> ::= $\underset{\text{<bound variable>}}{\overset{\text{<limit>}}{\square}}$ <guarded command>

<alternative command> ::= [<guarded command set>]

<repetitive command> ::= Σ <alternative command>

Examples. $[x \geq y \rightarrow m:=x \;\square\; y \geq x \rightarrow m:=y]$

Σ [c:character; west?c $\rightarrow$ east!c]

Σ [i < size; content(i) $\neq$ n $\rightarrow$ i:= i+1]

[n = content $\rightarrow$ skip

n < content $\rightarrow$ left! insert(n)

n > content $\rightarrow$ right! insert(n)

]

$\Sigma[\; \underset{i}{\overset{c}{\square}}$ X(i)?V( ) $\rightarrow$ val:=val+1

$\square \; \underset{i}{\overset{c}{\square}}$ val > 0; X(i)?P( ) $\rightarrow$ val:=val-1

]

An alternative command specifies execution of exactly one element of its guarded command set. If several of its guards are feasible, the choice between them is arbitrary. If none of the ready guards is feasible, the command is delayed until the first guard becomes ready and feasible; and the corresponding guarded command is selected. If all the guards are ready, but none of them are feasible, the alternative command fails.

A repetitive command specifies repeated execution of its alternative command. If all its guards are ready but none of them is feasible, the repetitive command is successfully completed, and no further repetition occurs; otherwise the alternative command is executed once, followed by repetition of the whole repetitive command.

A guarded array takes the form

$$\overset{n}{\underset{i}{\Box}} \quad G \to S$$

It denotes a set of n guarded commands, one for each value of i between 0 and n-1. It is equivalent to

$$G^i_o \to S^i_o \; [] \; G^i_1 \to S^i_1 \; [] \ldots [] \; G^i_{n-1} \to S^i_{n-1}$$

where $G^i_j \to S^i_j$ is the result of replacing all occurrences of the bound variable i in $G \to S$ by the numeral j.

2.6. Recursion.

<recursive command> ::= μ<identifier>: <command>

<recursive call> ::= <identifier>

A recursive command introduces an identifer as a name for the whole command. The scope of the identifier is confined to this command. Each occurrence of this identifier as a recursive call within the command is equivalent to a fresh copy of the whole command.

3. Coroutines.

This section contains simple examples of coroutines expressed as communicating sequential processes. The section heading gives the name of the fragment of program that it contains; and these names may be used in subsequent sections.

3.1. COPY

Problem: Copy character output by a process named west to a process named east.

Solution:

```
Σ[ west?c:character → east!c]
```

Notes: When the west process terminates, the input west?c will fail, causing termination of the repetitive command, and thus the COPY process also terminates. Any subsequent input command from east will therefore fail.

The COPY process acts as a single buffer, in that it permits west to work on production of its next character before east has accepted the previously produced one.

3.2. SQUASH

Problem: Adapt the program COPY to replace every pair of consecutive asterisks '**' by an upward arrow '↑'. You may assume that west does not terminate with an asterisk.

Solution:

```
Σ[west?c:character →
    [c ≠ asterisk → east!c
    [] c = asterisk → west?c;
        [ c ≠ asterisk → east!asterisk;east!c
        [] c = asterisk → east!upward arrow
        ]
      ]
    ]
```

Note: Since west does not end with asterisk, the second input west?c will not fail.

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:

```
                                          80
        Σ[cardreader ? cardimage :        Σ character  →

                        i:integer; i:=0;

          Σ [  i < 80 → X! cardimage(i); i:=i+1 ];    X!space

        ]
```

Note: $\overset{80}{\Sigma}$ character denotes an array of 80 characters with subscript bounds from 0 to 79.

3.4. ASSEMBLE

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

```
                  125
lineimage : Σ character;
i:integer; i:=0;
 Σ[X?c:character  →
           lineimage(i):=c;
            [i < 124 → i:=i+1
            []i = 124 → lineprinter!lineimage; i:=0
            ]
 ] ;
[i = 0  → skip
[]i > 0   →      Σ [ i < 125 → lineimage(i):=space; i:=i+1] ;
                     lineprinter!lineimage
 ]
```

3.5. REFORMAT

Problem: read a sequence of cards (80 characters each), and print their contents on a lineprinter (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 ]
```

3.6. Conway's example. [ ]

Problem: Adapt the program of 3.5. to replace every pair of consecutive asterisks by an upward arrow.

Solution:

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

3.7. BUFFER

Problem: Adapt the program of 3.3 so that it will proceed in parallel with operation of the physical card reader.

Solution: [cardreader: CARDCOPY || X: DISASSEMBLE]

where CARDCOPY =

[cardreader?cardimage: $\overset{80}{\Sigma}$ character $\rightarrow$ X! cardimage]

Note: Input by CARDCOPY from the cardreader will come from the global (physical) card reader; this is the effect of scope rule for process names. However, input by DISASSEMBLE from the cardreader will come from CARDCOPY, which is a more local process with the same name; this is a consequence of normal scope rules, as in ALGOL 60. Similarly, output from CARDCOPY to X will be directed to DISASSEMBLE, whereas output from DISASSEMBLE to X will go to a more global X.

3.8. DOUBLE BUFFER

Problem. Adapt the program of 3.7 to interpose an additional buffer to smooth temporary mismatch in speed between the physical card reader and the rest of the program.

Solution:

```
[ cardreader: CARDCOPY || X: BUFFER    ]
```

Note: A good trick can be played twice. But the necessary depth of the nesting begins to look unattractive. Perhaps a more specialised notation should be introduced for such chains of communicating processes.

4. Procedures, functions and classes.

This section illustrates the use of communicating sequential processes in place of procedures and functions- it also illustrates their role as representations of abstract data structures (Hoare), like the classes of SIMULA 67. Value and result parameters are passed by input and output; however, name parameters cannot be expressed at all.

Problem. To represent a procedure which inputs from its calling process X two integer parameters x and y, which then outputs to X two integer results, equal to x div y and x mod y. The function should be represented by a process, which can be used as many times as required, and which can compute in parallel with the calling process. The following solution is not intended to be efficient:

```
Solution:  Σ [ X?(x,y:integer); y > 0 →
                 quot,rem:integer; quot:=0; rem:=x;
             Σ [ rem ≥ y   →  rem:=rem-y;  quot:=quot+1 ];

                   X!(quot,rem)

               ]
           ]
```

Note: the guard y > 0 checks the validity of the parameter transmitted by X. The whole program will fail immediately if this guard is violated.

4.2. FACTORIAL.

Problem: to represent a procedure which inputs from its user X a non-negative integer n, and responds by outputting back to the user the value n! The algorithm used should be recursive rather than efficient.

```
μF:
  Σ[ X?n:integer   →
        [n = 0 → X!1

        []n > 0  →  [fac:F

                          || X :  [fac! n-1;  fac?m:integer; X! m x n]

               ]
        ]
  ]
```

Note: when n > 0, another instance of the process F is created, with name fac. This process will input n-1 and output its factorial to m. Note also that the output X! m x n, occurring inside a process named X, goes to the same global process which supplied the initial input X?n.

4.3. SMALLSET [15]

Problem: to represent a set of values of type basetype, where the size of the set never exceeds N. The set should be represented as a process which inputs instructions from its user process X, and outputs the result of its operations when required: its operations should proceed in parallel with its user wherever possible. There are two types of instruction from X, using constructors "insert" and "has" to distinguish them:

(1)	s!insert(n)	requests insertion of the value n in the set s
(2)	s!has(n);s?b	enquires whether n is a member of the set s. The result is assigned to the boolean variable b.

The initial value of the set should be empty.

Solution:

```
    content: $\overset{N}{\Sigma}$ basetype ; size:integer;  size:=0;

   *[ X?has(n:basetype) →

                               SEARCH;   X!(i ≥ size)

       □X?insert(n:basetype)        →

                           SEARCH;
                          [ i < size → skip
                          □ i = size; size < N     →

                                 content(size):=n; size:=size+1
                          ]
   ]
```

where SEARCH $=_{df}$

```
      i:integer; i:=0;

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

Note: the guard size < N will cause the alternative command (and hence the whole process) to fail if too many members are inserted.

4.4. TREESET.

Problem: to represent a set values of type basetype as a binary search tree. The external behaviour of the representation should be the same as for problem 4.3, but faster. Do not attempt to balance the tree.

Solution: We write a process to represent each node of the tree. Each node goes through one, two, or three of the following phases:

(1) to begin with, it represents the empty set, answering false to any membership enquiry.

(2) after the first insert, (if any), it represents a unit set, and answers membership enquiries by testing equality with its unique content.

(3) after the next insert (if any), it grows two branches, left and right, to which it passes on any instructions which it cannot execute directly. These branches are also nodes, with the same behaviour as that described above. Consequently, recursion is the appropriate program structure.

```
    μNODE:
(1)  [Σ[X?has(n:basetype) → X!false ];
(2)   X?insert(content:basetype);
     Σ[X?has(n:basetype) → X!(n=content)] ;
(3)  [left:NODE || right:NODE
     ||X:  Σ [X?insert(n:basetype)→

                  [n = content → skip
                  []n < content → left!insert(n)
                  []n > content →  right!insert(n)
                  ]

         []        X?has(n:basetype)

                 [n=content → X!true
                 []n < content → left!has(n)
                        left?b:boolean ;  X! b
                 []n > content → right! has(n)

                       right?b:boolean ; X!b
                 ]

         ]

     ]]
```

4.5. SEQUENCE;

Problem: to represent a sequence of integers as an array, assuming that the maximum length of the sequence never exceeds N. The sequence starts empty, and responds to instructions of the form:

```
seq!append(x)  : adds x to the end of the queue
seq!more( );[ seq?remove(x) →  ...

           [] seq?empty( )  →  ...

            ]
```

which either removes the first member of the sequence and assigns its value to x, or gives the answer "empty( )". The solution should use a cyclic buffer.

```
                     N
Solution:   content: Σ integer;
          Σ[in,out:integer; in:=0; out:=0;
            comment  out ≤ in ≤ out+N;

            Σ [  in < out+N; X?append(x:integer)  →

                       content(in mod N):=x; in:=in+1

          [] X?more( ) → [out < in → X! remove(buffer(out)); out:=out+1

                        [] out=in → X!empty( )

                         ]
               ]
          ]
```

Note: the guard in < out+N protects the sequence against overfilling. If it fails, the whole program fails.

5. Monitors and Conditional Critical Regions.

This section illustrates the solution of some problems in scheduling and synchronisation. The role of a monitor is taken by a sequential process, communicating with an array of client processes. Guards are used like conditions of a conditional critical region to postpone acceptance of undesirable inputs.

5.1. BOUNDED BUFFER.

Problem: to smooth a temporary mismatch of the speed at which a producer outputs portions and the speed of input by a consumer. This is achieved by use of a buffer containing at most N portions. The producer contains commands of the form:

```
            X!append(x)
```

which appends the value of x to the end of the buffer, and the consumer contains pairs of commands:

```
      X!more ( )    X?remove(p:portion)
```

which waits until the buffer is nonempty, and then removes the first element, assigning its value to p.

```
                       N
Solution:     buffer: Σ portion;
              in,out:integer;  in:=0; out:=0;
              comment  out ≤ in ≤ out+N;

             Σ [in < out+N;  prod?append(x:portion) → buffer(in mod N):=x; in:=in+1

             [] out < in,cons? more ( ) → cons! remove (buffer(out mod N));out:=out+1

              ]
```

Note: when out=in, the second guarded command will not be selected, even when the consumer asks for more. But when the producer outputs a portion, the first guarded command will be executed; and on the next repetition of the repetitive command the condition out < in will be true, and the delayed command from the consumer can be accepted.

5.2 Semaphore (Dijkstra)

Problem: to implement an integer semaphore, shared among an array X: $\prod_{i}^{c}$ client.

Each client increments the semaphore by an output command sem!V( ) and decrements it by sem!P( ).

Solution:

```
sem:  [ val:integer;  val:=0;  comment  val ≥ 0;
        Σ [ □ᶜᵢ X(i)?V( ) → val:val+1
            [] □ᶜᵢ val > 0;   X(i)?P( ) →   val:=val-1
          ]
      ]
```

Note: the guarded array provides an appropriate method for communicating with an arbitrary member of an array of processes. The value of the bound variable i indicates the identity of the process invoking the P or V operation. In the problem given above, no use is made of this information.

5.3. Multiple decrement

Problem: adapt the semaphore described above to input a parameter, indicating the number of units by which the value is to be changed. Subtraction must be delayed if it would make the value negative.

Solution:

```
                val:integer;  val:=0 →
        Σ [ □ᶜᵢ X(i)?V(n:integer) →  val:=val+n

          [] □ᶜᵢ X(i)?P(n:integer); val-n ≥ 0 → val:=val-n
          ]
```

Note: the condition val-n ≥ 0 ensures that val will not go negative. If X(i) attempts to output P(n), with an n that violates this condition, the guard will be infeasible, and the second guarded command will not be selected on the current iteration of the repetitive comand. The delay in rejection of the input until after inspection of the input value n is a startling but useful consequence of the definition of a guard; but it creates problems for implementation.

5.4 Explicit scheduling

Problem: A set of R resources numbered 0 to R-1 is to be shared among an array $X: \overset{C}{\underset{i}{\|}}$ user. Each user acquires at most one resource at a time. A resource is acquired by two commands:

```
allocator!request( ); allocator?whichone(r)
```

of which the second assigns to r the number of the acquired resource.

The resource r is released by:

```
allocator!release(r)
```

Resources are to be allocated on the basis of "first come, first served". The solution may use the SEQUENCE process defined in 4.5 to represent both the set of free resources and the queue of waiting processes.

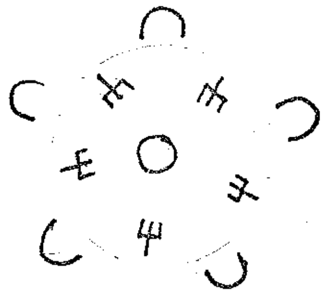
```
Solution [  free : SEQUENCE || queue:SEQUENCE
         || X: [r:integer;  r:=0;   Σ [r < R → free!append(r);r:r+1];


   Σ [ $\overset{C}{\underset{i}{\square}}$ X(i)?request( ) → queue!append(i); free!more( );
              [  free?empty( ) → skip
              [] free?remove(r:integer) → queue!more( )
                     queue?remove(f:integer); X(f)!whichone(r)
              ]
   [] $\overset{C}{\underset{i}{\square}}$ X(i)?release(r:integer) → free!append(r);queue!more( ) ;
              [queue?empty( ) → skip
              [] queue?remove(f:integer) → free!more( )
                      free?remove(r)  ;  X(f)!whichone(r)
              ]
   ]]
```

Note: if a sequence is known to be nonempty, there is no need to test for this case.

5.6. Dining philosophers (problem and solution 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. You may assume that there is only space for four philosophers in the dining room.

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

```
PHIL =
      Σ [ 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 =
      Σ[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;
          5
    Σ [  □ occupancy < 4;  phil(i)?enter( )
         i
                          → occupancy:=occupancy+1
      ]
```

Allthese components operate in parallel:

```
                  4                     4
[ room:ROOM  || fork: ∏  FORK || phil:  ∏  PHIL ]
                  i                     i
```

Conclusion.

The examples of the previous sections have shown a wide range of application of a simple form of input, output, and parallelism. Their combination with an extended form of Dijkstra's guard gives some surprising results.

It is therefore tempting to explore yet further extensions to the concept of a guard. For example, if an output command were permitted as a guard element, there would be no need for the extra commands "X!more( )" in the problems 4.5, 5.1, and 5.4; and their solutions would be somewhat neater. But it is doubtful whether this neatness would extend to more realistic examples; and the additional problems of efficient implementation would probably be severe.

An even more surprising extension would be to permit any command whatsoever to appear as a guard element. This would permit the definition of Randell's recovery block, eg.

```
[main block; acceptance test → skip
□alternate; acceptance test  →skip
]
```

However, even with specially designed hardware, the problems of implementation in the presence of input and output and nontermination are severe; and could lead to an uncontrolled loss of efficiency.

The language proposal described in this paper is highly speculative, and needs much further investigation before it can be recommended for implementation or widespread use. Such investigation is more likely to reveal a need to restrict the generality of the proposal rather than to extend it.

Acknowledgements.

This paper owes its inspiration to E.W. Dijkstra (1975), and has benefitted by helpful suggestions by D. Gries, D.Q.M. Fay, and N. Wirth.

REFERENCES:

1. Naur, Peter (ed.), Report on the Algorithmic Language ALGOL 60, Comm. ACM 3, (May 1960), 299 - 314.

2. ISO standard input/output.

3. FORTRAN

4. PL/I

5. UNIX

6. SIMULA 67

7. Brinch Hansen, P. Concurrent PASCAL.

8. Liskov, B.H. CLU.

9. Woolf, W.

10. v. Wijngaarden, A.

11. Dijkstra, E.W. Co-operating Sequential Processes.

12. Hoare, C.A.R. Towards a Theory of Parallel Programming, in Operating Systems Techniques. Academic Press, pp.

13. Dijkstra, E.W. Guarded Commands, Nondeterminacy, and formal derivation of programs. Comm. ACM, 18, 8 (Aug.1975), pp. 453 - 457.

14. Conway, M.E. Design of a Separable Transition-Diagram Compiler. Comm. ACM 6, 7 ( ) pp. 396 - 408.

15. Hoare, C.A.R. Proof of correctness of Data Representations. Acta Informatica.

16. Dijkstra, E.W. Operating Systems Techniques, in Academic Press.

17. Dijkstra, E.W. Private communication.