COMMUNICATING SEQUENTIAL PROCESSES

C. A. R. HOARE

(1978)

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.

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. They are often added to a programming language only as an afterthought.

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 a semicolon). Less agreement has been reached about the design of other important program structures, and many suggestions have

C. A. R. Hoare, Communicating sequential processes. *Communications of the ACM 21*, 8 (August) 1978, 666–677. Copyright © 1978, Association for Computing Machinery, Inc. Reprinted by permission.

been made: Subroutines (Fortran), procedures (Algol 60 (Naur 1960)), entries (PL/I), coroutines (UNIX (Thompson 1976)), classes (SIMULA 67 (Dahl et al. 1967)), processes and monitors (Concurrent Pascal (Brinch Hansen 1975)), clusters (CLU (Liskov 1974)), forms (ALPHARD (Wulf et al. 1976)), actors (Atkinson and Hewitt 1976).

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 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 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 synchronize with each other. 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 (van Wijngaarden 1969), 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 some technologies of hardware implementation. A greater variety of methods has been proposed for synchronization: semaphores (Dijkstra 1968), events (PL/I), conditional critical regions (Hoare 1972a), monitors and queues (Concurrent Pascal (Brinch Hansen 1975)), and path expressions (Campbell 1974). Most of these are demonstrably adequate for their purpose, but there is no widely recognized 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 (1975a) are adopted (with a slight change of notation) as sequential control structures, and as the sole means of introducing and controlling nondeterminism.

(2) A parallel command, based on Dijkstra's *parbegin* (1968), specifies concurrent execution of its constituent sequential commands (processes). All the processes start simultaneously, and the parallel command ends only when

they are all finished. They may not communicate with each other by updating global variables.
(3) Simple forms of input and output command are introduced. They are used for communication between concurrent processes.
(4) Such communication occurs when one process names another as destination for output *and* 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 *no* 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 process.
(5) Input commands may appear in guards. A guarded command with an input guard is selected for execution only if and when the source named in the input command is ready to execute the corresponding output command. If several input guards of a set of alternatives have ready destinations, only one is selected and the others have *no* effect; but the choice between them is arbitrary. In an efficient implementation, an output command which has been ready for a long time should be favoured; but the definition of a language cannot specify this since the relative speed of execution of the processes is undefined.
(6) A repetitive command may have input guards. If all the sources named by them have terminated, then the repetitive command also terminates.
(7) A simple pattern-matching feature, similar to that of Reynolds (1965), is used to discriminate the structure of an input message, and to access its components in a secure fashion. This feature is used to inhibit input of messages that do not match the specified pattern.

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 optimizations are appropriate in the different cases). It is consequently 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. In other respects also, the language has been stripped to the barest minimum necessary for explanation of its more novel features.

The concept of a communicating sequential process is shown in Sections 3–5 to provide a method of expressing solutions to many simple programming exercises which have previously been employed to illustrate the use of various proposed programming-language features. This suggests that the process may constitute a synthesis of a number of familar and new programming ideas. The reader is invited to skip the examples which do not interest him.

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 problems of efficient implementation, which may be particularly serious 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) re-introduction of distinctive notations for the most common and useful special cases; (3) development of automatic optimization techniques; and (4) the design of 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 to the problems tackled. Further discussion of these and other points will be found in Section 7.

2 Concepts and notations

The style of the following description is borrowed from Algol 60 (Naur 1960). Types, declarations, and expressions have not been treated; in the examples, a Pascal-like notation (Wirth 1971) has usually been adopted. The curly braces { } have been introduced into BNF to denote none or more repetitions of the enclosed material. (Sentences in parentheses refer to an implementation: they are not strictly part of a language definition.)

```
<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. It may succeed or fail. Execution of a simple command, if successful, may have an effect on the internal state of the executing device (in the case of assignment), or on its external environment (in the case of output), or on both (in the case of input). Execution of a structured command involves execution of some or all of its constituent commands, and if any of these fail, so does the structured command. (In this case, whenever possible, an implementation should provide some kind of comprehensible error diagnostic message.)

A null command has no effect and never fails.

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

2.1 Parallel commands

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

Each process of a parallel command must be *disjoint* from every other process of the command, in the sense that it does not mention any variable which occurs as a target variable (see Sections 2.2 and 2.3) in any other process.

A process label without subscripts, or one whose label subscripts are all integer constants, serves as a name for the command list to which it is prefixed; its scope extends over the whole of the parallel command. A process whose label subscripts include one or more ranges stands for a series of processes, each with the same label and command list, except that each has a different combination of values substituted for the bound variables. These values range between the lower bound and the upper bound inclusively. For example, `X(i:1..n) :: CL` stands for

$\texttt{X(1) :: CL}_1 \| \texttt{X(2) :: CL}_2 \| \ldots \| \texttt{X(n) :: CL}_\texttt{n}$

where each $\texttt{CL}_j$ is formed from `CL` by replacing every occurrence of the bound variable `i` by the numeral `j`. After all such expansions, each process label in a parallel command must occur only once and the processes must be well formed and disjoint.

A parallel command specifies concurrent execution of its constituent processes. They all start simultaneously and the parallel command terminates successfully only if and when they have all successfully terminated. The relative speed with which they are executed is arbitrary.

Examples:

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

Performs the two constituent commands in parallel, and terminates only when both operations are complete. The time taken may be as low as the longer of the times taken by each constituent process, i.e. the sum of its computing, waiting, and transfer times.

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

The three processes have the names `west`, `X`, and `east`. The capitalized 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 specifed by the command list `ROOM`. The behaviour of the five processes `fork(0)`, `fork(1)`, `fork(2)`, `fork(3)`, `fork(4)`, is specifed by the command list `FORK`, within which the bound variable `i` indicates the identity of the particular fork. Similar remarks apply to the five processes `PHIL`.

2.2 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 expression denotes a value which is computed by an executing device by application of its constituent operators to the specified operands. The value of an expression is undefined if any of these operations are undefined. The value denoted by a simple expression may be simple or structured. The value denoted by a structured expression is structured; its constructor is that of the expression, and its components are the list of values denoted by the constituent expressions of the expression list.

An assignment command specifies evaluation of its expression, and assignment of the denoted value to the target variable. A simple target variable may have assigned to it a simple or a structured value. A structured target variable may have assigned to it a structured value, with the same constructor. The effect of such assignment is to assign to each constituent simpler variable of the structured target the value of the corresponding component of the structured value. Consequently, the value denoted by the target variable, if evaluated *after* a successful assignment, is the same as the value denoted by the expression, as evaluated *before* the assignment.

An assignment fails if the value of its expression is undefined, or if that value does not *match* the target variable, in the following sense: 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. A structured value with no components is known as a "signal".

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)	`x:= cons(left, right)`	constructs a structured value and assigns it to x.
(4)	`cons(left, right) := x`	fails if x does not have the form `cons(y, z)`; but if it does, then y is assigned to left, and z is assigned to right.
(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.

Note: Successful execution of both (3) and (4) ensures the truth of the postcondition `x = cons(left, right)`; but (3) does so by changing x and (4) does so by changing left and right. Example (4) will fail if there is *no* value of left and right which satisfes the postcondition.

2.3 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 two concurrently operating sequential processes. Such a process may be implemented in hardware as a special-purpose device (e.g. cardreader or line printer), or its behaviour may be specified by one of the constituent processes of a parallel command. Communication occurs between two processes of a parallel command whenever (1) an input command in one process specifies as its source the process name of the other process; (2) an output command in the other process specifies as its destination the process name of the first process; and (3) the target variable of the input command matches the value denoted by the expression of the output command. On these conditions, the input and output commands are said to *correspond.* Commands which correspond are executed simultaneously, and their combined effect is to assign the value of the expression of the output command to the target variable of the input command.

An input command fails if its source is terminated. An output command fails if its destination is terminated or if its expression is undefined.

(The requirement of synchronization of input and output commands means that an implementation will have to delay whichever of the two commands happens to be ready first. The delay is ended when the corresponding command in the other process is also ready, or when the other process terminates. In the latter case the first command fails. It is also possible that the delay will never be ended, for example, if a group of processes are attempting communication but none of their input and output commands correspond with each other. This form of failure is known as a deadlock.)

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 a pair of values and assign them to `x` and `y`.
(4)	`DIV!(3*a + b, 13)`	to process `DIV`, output the two specifed 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)` (≡ `x:= 3*a + b; y:= 13`).

(5)	`console(i)?c`	from the `i`th element of an array of consoles, input a value and assign it to `c`.
(6)	`console(j - 1)!"A"`	to the `(j - 1)`th console, output character ''A''.
(7)	`X(i)?V()`	from the `i`th of an array of processes `X`, input a signal `V()`; refuse to input any other signal.
(8)	`sem!P()`	to sem output a signal `P()`.

2.4 Alternative and repetitive commands

<repetitive command> ::= ∗ <alternative command>
<alternative command> ::= [<guarded command>
    {▯<guarded command>}]
<guarded command> ::= <guard>→<command list>
    |(<range>{,<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 guarded command with one or more ranges stands for a series of guarded commands, each with the same guard and command list, except that each has a different combination of values substituted for the bound variables. The values range between the lower bound and upper bound inclusive. For example, `(i:1..n)G` $\rightarrow$ `CL` stands for

$G_1 \rightarrow CL_1 \square G_2 \rightarrow CL_2 \square \ldots \square G_n \rightarrow CL_n \square$

where each $G_j \rightarrow CL_j$ is formed from $G \rightarrow CL$ by replacing every occurrence of the bound variable `i` by the numeral `j`.

A guarded command is executed only if and when the execution of its guard does not fail. First its guard is executed and then its command list. A guard is executed by execution of its constituent elements from left to right. A Boolean expression is evaluated: If it denotes false, the guard fails, but an expression that denotes true has no effect. A declaration introduces a fresh variable with a scope that extends from the declaration to the end of the guarded command. An input command at the end of a guard is executed only if and when a corresponding output command is executed. (An implementation may test whether a guard fails simply by trying to execute it, and discontinuing execution if and when it fails. This is valid because such a discontinued execution has no effect on the state of the executing device.)

An alternative command specifies execution of exactly one of its constituent guarded commands. Consequently, if all guards fail, the alternative command fails. Otherwise an arbitrary one with successfully executable guard is selected and executed. (An implementation should take advantage of its freedom of selection to ensure efficient execution and good response. For example, when input commands appear as guards, the command which corresponds to the earliest ready and matching output command should in general be preferred; and certainly, no executable and ready output command should be passed over unreasonably often.)

A repetitive command specifies as many iterations as possible of its constituent alternative command. Consequently, when all guards fail, the repetitive command terminates with no effect. Otherwise, the alternative command is executed once and then the whole repetitive command is executed again. (Consider a repetitive command when all its true guard lists end in an input guard. Such a command may have to be delayed until either (1) an output command corresponding to one of the input guards becomes ready, or (2) all the sources named by the input guards have terminated. In case (2), the repetitive command terminates. If neither event ever occurs, the process fails (in deadlock).)

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.

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

This command inputs repeatedly from any of ten consoles, provided that the corresponding element of the Boolean array continue is true. The bound variable i identifies the originating console. Its value, together with the character just input, is output to X, and an acknowledgment signal is sent back to the originating console. If the character indicated `sign off`, `continue(i)` is set false, to prevent further input from that console. The repetitive command terminates when all ten elements of continue are false. (An implementation should ensure that no console which is ready to provide input will be ignored unreasonably often.)

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

(Here, and elsewhere, capitalized words `INSERT` and `SEARCH` stand as abbreviations for program text defined separately.)

On each iteration this command accepts from X *either* (a) a request to `insert(n)`, (followed by INSERT) *or* (b) a question `has(n)`, to which it outputs an answer back to X. The choice between (a) and (b) is made by the next output command in X. The repetitive command terminates when X does. If X sends a non-matching message, deadlock will result.

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

On each iteration, accept *either* a `V()` signal from X and increment val, *or* 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 when X is terminated and val ≤ 0.

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 $\uparrow$. Assume that the final 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 cardfile 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; cardfile?cardimage →
    i:integer; i:= 1;
    *[i≤80 → X!cardimage(i); i:= i + 1]
    X!space
]
```

Notes: (1) `(1..80)character` declares an array of 80 characters, with subscripts ranging between 1 and 80. (2) The repetitive command terminates when the cardfile process terminates.

3.4 Assemble

Problem: To read a stream of characters from 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: When X terminates, so will the first repetitive command of this process. The last line will then be printed, if it has any characters.

3.5 Reformat

Problem: Read a sequence of cards of 80 characters each, and print the characters on a line printer at 125 characters per line. Every card should be

followed by an extra space, and last line should be completed with space if necessary.
Solution:

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

Notes: (1) The capitalized names stand for program text defined in previous sections. (2) The parallel command is designed to terminate after the card file has terminated. (3) This elementary problem is difficult to solve elegantly without coroutines.

3.6 Conway's problem (1963)

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 nonrecursive subroutine can be readily implemented as a coroutine, provided that (1) its parameters are called "by value" and "by result", and (2) it is disjoint from its calling program. Like a Fortran subroutine, a coroutine may retain the values of local variables (*own* 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 subroutine will terminate when its user does. 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 (Hoare 1972b), will also contain a repetitive command which represents each entry by an alternative input to a structured target with the entry name as constructor. For example,

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

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 simulated by an array of processes, one for each level of recursion. The user 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.

This clumsy simulation of recursion would be even more clumsy for a mutually recursive algorithm. It would not be recommended for conventional programming; it may be more suitable for an array of microprocessors for which the fixed upper bound is also realistic.

In this section, we assume each subroutine is used only by a *single* user process (which may, of course, itself contain parallel commands).

4.1 Function: division with remainder

Problem: Construct a process to represent a function-type subroutine, which accepts a positive dividend and divisor, 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
]
```

4.2 Recursion: factorial

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

```
[fac(i:1..limit)::
*[n:integer;fact(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
]
```

Note: This unrealistic example introduces the technique of the "iterative array" which will be used to better effect in later examples.

4.3 Data representation: small set of integers (Hoare 1972b)

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 X: (1) S!insert(n), insert the integer n in the set, and (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 the previous insertion is complete.

4.4 Scanning a 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 a repetitive command of the form:

```
S!scan(); more:boolean; more:= true;
*[more;x:integer; S?next(x) → ... deal with x ...
□more; S?noneleft() → more:= false
]
```

where `S!scan()` sets the representation into a scanning mode. The repetitive command 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 no members 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 repetitive command of `S`:

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

4.5 Recursive data representation: small set of integers

Problem: Same as above, but an array of processes is to be used to achieve a high degree of parallelism. Each process should contain at most one number. When it contains no number, it should answer *false* to all inquiries 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(0)`. For efficiency, the set should be sorted, i.e. the `i`th process should contain the `i`th largest number.

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 user process `S(0)` inquires 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) All repetitive commands and all processes of the array will terminate after the user process `S(0)` terminates

4.6 Multiple exits: remove the least member

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

```
S(1)!least(); [x:integer;S(1)?x → deal with x ...
               ▯S(1)?noneleft() → ...
               ]
```

or, if he wishes to scan and empty the set, he may write:

```
S(1)!least();more:boolean; more:= true;
  *[more; x:integer; S(1)?x → ... deal with x ...; S(1)!least()
  ▯more; S(1)?noneleft() → more:= false
  ]
```

Hint: Introduce a Boolean variable, `b`, initialized 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 shows how a monitor can be regarded as a single process which communicates with more than one user process. However, 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. For example: `*[(i:1..100)X(i)?(value parameters)` $\rightarrow$ `... ;X(i)!(results)]`. Here, the bound variable `i` is used to send the results back to the calling process. 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. For example, two successive inputs from the same process are inhibited by `j = 0; *[(i:1..100)i` $\neq$ `j; X(i)?(values)` $\rightarrow$ `...; j:= i]`. Any attempted output from `X(j)` will be delayed until a subsequent iteration, after the output of some process `X(i)` has been accepted and dealt with.

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 1972a) and it obviates the need for special synchronizing variables such as events, queues, or conditions. However, the absence of these special facilities certainly makes it more difficult or less efficient to solve problems involving priorities—for example, the scheduling of head movement on a disk.

5.1 Bounded buffer

Problem: Construct a buffering process `X` to smooth variations in the speed of output of portions by 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 buffer is empty and 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 iteration. (3) Similar remarks apply to the producer, when in = out + 10. (4) X is designed to terminate when out = in and the producer has terminated.

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
  ]
```

Notes: (1) In this process, no use is made of knowledge of the subscript i of the calling process. (2) The semaphore terminates only when all hundred processes of the process array X have terminated.

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 (see Figure 1). 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.

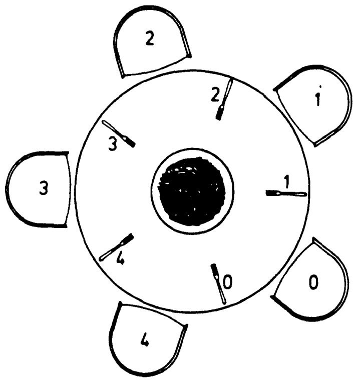

Fig. 1

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

```
PHIL = *[... during ith lifetime ... →
        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 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]
```

Notes: (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. Hint: Prevent 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 in the second example, even the question of termination is ignored.

6.1 Prime numbers: the sieve of Eratosthenes (McIlroy 1968)

Problem: To print in ascending order all primes less than 10000. 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 are multiples 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 < 10000 → 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 (Dijkstra 1972, pp. 27–32).

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. The solution should take the form shown in Figure 2. Each of the nine nonborder nodes inputs a vector component from the west and a partial sum from the north. Each node 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 zeros and the east border is just a sink. No provision need be made for termination nor for changing the values of the array `A`.

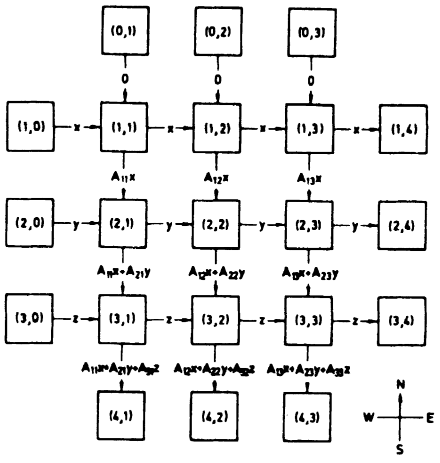

Fig. 2

Solution: 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 the remaining processes are:

```
NORTH = *[true → M(1,j)!0]
EAST = *[x:real; M(i,3)?x → skip]
CENTER = *[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)
  ]
```

7 Discussion

A design for a programming language must necessarily involve a number of decisions which seem to be fairly arbitrary. The discussion of this section is intended to explain some of the underlying motivation and to mention some unresolved questions.

7.1 Notations

I have chosen single-character notations (e.g. !,?) to express the primitive concepts, rather than the more traditional boldface or underlined English words. As a result, the examples have an APL-like brevity, which some readers find distasteful. My excuse is that (in contrast to APL) there are only a very few primitive concepts and that it is standard practice of mathematics (and also good coding practice) to denote common primitive concepts by brief notations (e.g. $+, \times$). When read aloud, these are replaced by words (e.g. plus, times).

Some readers have suggested the use of assignment notation for input and output:

<target variable> := <source>
<destination> := <expression>

I find this suggestion misleading: it is better to regard input and output as distinct primitives, justifying distinct notations.

I have used the same pair of brackets (`[...]`) to bracket all program structures, instead of the more familiar variety of brackets (**if**..**fi**, **begin**..**end**, **case**..**esac**, etc.). In this I follow normal mathematical practice, but I must also confess to a distaste for the pronunciation of words like **fi**, **od**, or **esac**.

I am dissatisfed with the fact that my notation gives the same syntax for a structured expression and a subscripted variable. Perhaps tags should be distinguished from other identifers by a special symbol (say #).

I was tempted to introduce an abbreviation for combined declaration and input, e.g. `X?(n:integer)` for `n:integer; X?n`.

7.2 Explicit naming

My design insists that every input or output command must name its source or destination explicitly. This makes it inconvenient to write a library of processes which can be included in subsequent programs, independent of the process names used in that program. A partial solution to this problem is to allow one process (the *main* process) of a parallel command to have an empty label, and to allow the other processes in the command to use the empty process name as source or destination of input or output.

For construction of large programs, some more general technique will also be necessary. This should at least permit substitution of program text for names defined elsewhere—a technique which has been used informally throughout this paper. The Cobol COPY verb also permits a substitution for formal parameters within the copied text. But whatever facility is introduced, I would recommend the following principle: Every program, after assembly with its library routines, should be printable as a text expressed wholly in the language, and it is this printed text which should describe the execution of the program, independent of which parts were drawn from a library.

Since I did not intend to design a complete language, I have ignored the problem of libraries in order to concentrate on the essential semantic concepts of the program which is actually executed.

7.3 Port names

An alternative to explicit naming of source and destination would be to name a *port* through which communication is to take place. The port names would be local to the processes, and the manner in which pairs of ports are to be connected by channels could be declared in the head of a parallel command.

This is an attractive alternative which could be designed to introduce a useful degree of syntactically checkable redundancy. But it is semantically equivalent to the present proposal, provided that each port is connected to exactly one other port in another process. In this case each channel can be identifed with a tag, together with the name of the process at the other end. Since I wish to concentrate on semantics, I preferred in this paper to use the simplest and most direct notation, and to avoid raising questions about the possibility of connecting more than two ports by a single channel.

7.4 Automatic buffering

As an alternative to synchronization of input and output, it is often proposed that an outputting process should be allowed to proceed even when the inputting process is not yet ready to accept the output. An implementation would be expected automatically to interpose a chain of buffers to hold output messages that have not yet been input.

I have deliberately rejected this alternative, for two reasons: (1) It is less realistic to implement in multiple disjoint processors, and (2) when buffering is required on a particular channel, it can readily be specified using the given primitives. Of course, it could be argued equally well that synchronization can be specifed when required by using a pair of buffered input and output commands.

7.5 Unbounded process activation

The notation for an array of processes permits the same program text (like an Algol recursive procedure) to have many simultaneous "activations"; however, the exact number must be specifed in advance. In a conventional single-processor implementation, this can lead to inconvenience and wastefulness, similar to the fixed-length array of Fortran. It would therefore be attractive to allow a process array with no a priori bound on the number of elements; and to specify that the exact number of elements required for a particular execution of the program should be determined dynamically, like the maximum depth of recursion of an Algol procedure or the number of iterations of a repetitive command.

However, it is a good principle that every actual run of a program with unbounded arrays should be identical to the run of some program with all its arrays bounded in advance. Thus the unbounded program should be defined as the "limit" (in some sense) of a series of bounded programs with increasing bounds. I have chosen to concentrate on the semantics of the bounded case—which is necessary anyway and which is more realistic for implementation on multiple microprocessors.

7.6 Fairness

Consider the parallel command:

```
[X::Y!stop()||Y::continue:boolean;continue:= true;
    *[continue; X?stop() → continue := false
      □continue → n:= n + 1
      ]
]
```

If the implementation always prefers the second alternative in the repetitive command of Y, it is said to be *unfair*, because although the output command in X could have been executed on an infinite number of occasions, it is in fact always passed over.

The question arises: Should a programming language definition specify that an implementation must be *fair*? Here, I am fairly sure that the answer is NO. Otherwise, the implementation would be obliged to successfully complete the example program shown above, in spite of the fact that its nondeterminism is unbounded. I would therefore suggest that it is the programmer's responsibility to prove that his program terminates correctly—without relying on the assumption of fairness in the implementation. Thus the program shown above is incorrect, since its termination cannot be proved.

Nevertheless, I suggest that an efficient implementation should try to be reasonably fair and should ensure that an output command is not delayed unreasonably often after it first becomes executable. But a proof of correctness must not rely on this property of an efficient implementation. Consider the following analogy with a sequential program: An efficient implementation of an alternative command tends to favour the alternative which can be most efficiently executed, but the programmer must ensure that the logical correctness of his program does not depend on this property of his implementation.

This method of avoiding the problem of fairness does not apply to programs such as operating systems which are intended to run forever, because in this case termination proofs are not relevant. But I wonder whether it is ever advisable to write or to execute such programs. Even an operating system should be designed to bring itself to an orderly conclusion reasonably soon after it inputs a message instructing it to do so. Otherwise, the only way to stop it is to "crash" it.

7.7 Functional coroutines

It is interesting to compare the processes described here with those proposed in Kahn (1974); the differences are most striking. There, coroutines are strictly deterministic: No choice is given between alternative sources of input. The output commands are automatically buffered to any required degree. The output of one process can be automatically fanned out to any number of processes (including itself!) which can consume it at differing rates. Finally, the processes there are designed to run forever, whereas my proposed parallel command is normally intended to terminate. The design in Kahn (1974) is based on an elegant theory which permits proof of the properties of programs. These differences are not accidental—they seem to be natural consequences of the difference between the more abstract applicative (or functional) approach to programming and the more machine-oriented imperative (or procedural) approach, which is taken by communicating sequential processes.

7.8 Output guards

Since input commands may appear in guards, it seems more symmetric to permit output commands as well. This would allow an obvious and useful simplification in some of the example programs, for example, in the bounded buffer (5.1). Perhaps a more convincing reason would be to ensure that the externally visible effect and behaviour of every parallel command can be modelled by some sequential command. In order to model the parallel command

```
Z::[X!2||Y!3]
```

we need to be able to write the sequential alternative command:

```
Z::[X!2 → Y!3□Y!3 → X!2]
```

Note that this cannot be done by the command

```
Z::[true → X!2; Y!3□true → Y!3; X!2]
```

which can fail if the process Z happens to choose the first alternative, but the processes Y and X are synchronized with each other in such a way that Y must input from Z before X does, e.g.

```
 Y::Z?y; X!go()
||X::Y?go();Z?x
```

7.9 Restriction: Repetitive command with input guard

In proposing an unfamiliar programming-language feature, it seems wiser at first to specify a highly restrictive version rather than to propose extensions—especially when the language feature claims to be primitive. For example, it is clear that the multidimensional process array is not primitive, since it can readily be constructed in a language which permits only single-dimensional arrays. But I have a rather more serious misgiving about the repetitive command with input guards.

The automatic termination of a repetitive command on termination of the sources of all its input guards is an extremely powerful and convenient feature but it also involves some subtlety of specification to ensure that it is implementable; and it is certainly not primitive, since the required effect can be achieved (with considerable inconvenience) by explicit exchange of `end()` signals. For example, the subroutine `DIV` (4.1) could be rewritten:

```
  [DIV::continue:boolean; continue:= true;
  *[continue; X?end() → continue:= false
  []continue; x,y:integer; X?(x,y) → ...; X!(quot,rem)
||X::USER PROG; DIV!end()
  ]
```

Other examples would be even more inconvenient.

But the dangers of convenient facilities are notorious. For example the repetitive commands with input guards may tempt the programmer to write them without making adequate plans for their termination; and if it turns out that the automatic termination is unsatisfactory, reprogramming for explicit termination will involve severe changes, affecting even the interfaces between the processes.

8 Conclusion

This paper has suggested that input, output, and concurrency should be regarded as primitives of programming, which underlie many familiar and less familiar programming concepts. However, it would be unjustified to conclude that these primitives can wholly replace the other concepts in a programming language. Where a more elaborate construction (such as a procedure or monitor) is frequently useful, has properties which are more simply provable, and can be implemented more efficiently than the general case, there is a strong reason 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.

Acknowledgments. 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 (1975b), and the paper has been improved in presentation and content by valuable and painstaking advice from D. Gries, D. Q. M. Fay, E. W. Dijkstra, N. Wirth, R. Milne, M. K. Harper, and its referees. The role of IFIP W.G.2.3 as a forum for presentation and discussion is acknowledged with pleasure and gratitude.

References

Atkinson, R., and Hewitt, C. 1976. Synchronisation in actor systems. Working Paper 83, M.I.T., Cambridge, Mass., Nov.

Brinch Hansen, P. 1975. The programming language Concurrent Pascal. *IEEE Trans. Software Eng. 1*, 2 (June), 199–207.

Campbell, R.H., and Habermann, A.N. 1974. The specification of process synchronisation by path expressions. *Lecture Notes in Computer Science 16*, Springer, 89–102.

Conway, M.E. 1963. Design of a separable transition-diagram compiler. *Comm. ACM 6*, 7 (July), 396–408.

Dahl, O.-J., et al. 1967. SIMULA 67, common base language. Norwegian Computing Centre, Forskningveien, Oslo.

Dijkstra, E.W. 1968. Co-operating sequential processes. In *Programming Languages*, F. Genuys, Ed., Academic Press, New York, 43–112.

Dijkstra, E.W. 1972. Notes on structured programming. In *Structured Programming*, Academic Press, New York, 1–82.

Dijkstra, E.W. 1975a. Guarded commands, nondeterminacy, and formal derivation of programs. *Comm. ACM 18*, 8 (Aug.), 453–457.

Dijkstra, E.W. 1975b. Verbal communication, Marktoberdorf, Aug.

Hoare, C.A.R. 1972a. Towards a theory of parallel programming. In *Operating Systems Techniques*, Academic Press, New York, 61–71.

Hoare, C.A.R. 1972b. Proof of correctness of data representations. *Acta Informatica 1*, 4, 271–281.

Kahn, G. 1974. The semantics of a simple language for parallel programming. In *Proc. IFIP Congress 74*, North Holland.

Liskov, B.H. 1974. A note on CLU. Computation Structures Group Memo. 112, M.I.T., Cambridge, Mass.

McIlroy, M.D. 1968. Coroutines. Bell Laboratories, Murray Hill, N.J.

Naur, P., Ed. 1960. Report on the algorithmic language ALGOL 60. *Comm. ACM 3*, 5 (May), 299–314.

Reynolds, J.C. 1965. COGENT. ANL-7022, Argonne Nat. Lab., Argonne, Ill.

Thompson, K. 1976. The UNIX command language. In *Structured Programming*, Infotech, Nicholson House, Maidenhead, England, 375–384.

van Wijngaarden, A., Ed. 1969. Report on the algorithmic language ALGOL 68. *Numer. Math. 14*, 79–218.

Wulf, W.A., London, R.L., and Shaw, M. 1976. Abstraction and verification in ALPHARD. Dept. of Comptr. Sci., Carnegie-Mellon U., Pittsburgh, Pa., June.

Wirth, N. 1971. The programming language PASCAL. *Acta Informatica 1*, 1, 35–63.