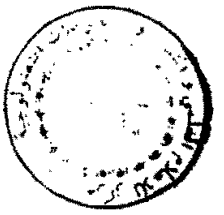
COMMUNICATING SEQUENTIAL PROCESSES.

C.A.R. HOARE.



Programming Techniques

S. L. Graham, R. L. Rivest Editors

Communicating Sequential Processes

C.A.R. Hoare
The Queen's University
Belfast, Northern Ireland

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

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

CR Categories: 4.20, 4.22, 4.32

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 modeled 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 been made: Subroutines (Fortran), procedures (Algol 60 [15]), entries (PL/I), coroutines (UNIX [17]), classes (SIMULA 67 [5]), processes and monitors (Concurrent Pascal [2]), clusters (CLU [13]), forms (ALPHARD [19]), actors (Hewitt [1]).

The traditional stored program digital computer has been designed primarily for deterministic execution of a single sequential program. Where the desire for greater speed has led to the introduction of parallelism, every attempt has been made to disguise this fact from the programmer, either by hardware itself (as in the multiple function units of the CDC 6600) or by the software (as in an I/O control package, or a multiprogrammed operating system). However, developments of processor technology suggest that a multiprocessor machine, constructed from a number of similar self-contained processors (each with its own store), may become 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 [18], 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 [6], events (PL/I), conditional critical regions [10], monitors and queues (Concurrent Pascal [2]), and path expressions [3]. Most of these are demonstrably adequate for their purpose, but there is no widely 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 [8] 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* [6], 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.

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

Author's present address: Programming Research Group, 45, Banbury Road, Oxford, England.

(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 favored; 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 [16], 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 familiar 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) reintroduction 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 [15]. Types, declarations, and expressions have not been treated; in the examples, a Pascal-like notation [20] 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 behavior 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>
```