PL-Circle: Paper Reading Group

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 programming, 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.

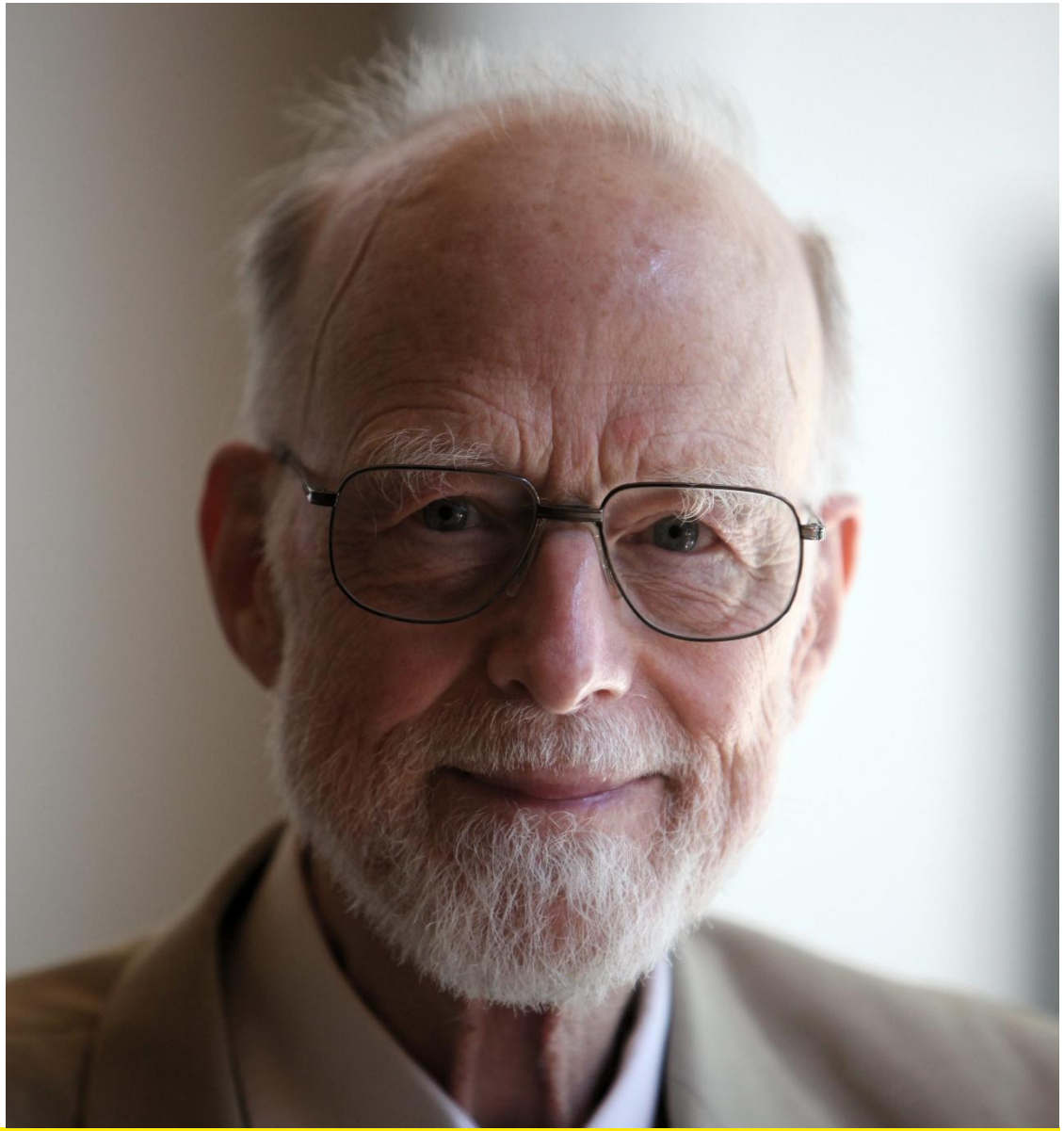
(3) Simple forms of input and output command are introduced. They are used for communication between concurrent processes.

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.

Hintergrund

- 1978
  - CSP wurde 1985 in Buch weiterentwickelt
- Charles Antony Richard (C.A.R.) Hoare
  - 1934
  - Britischer Informatiker
  - Queen’s University Belfast, University of Oxford
  - Quicksort
  - Hoare-Logik (Korrektheit von Programmen)



Syntax (1)

“Input and Output as Basic Primitives”

Input und Output Commands als Alternative zu Shared Memory

Ich habe eine Frage

Input (Lesen von anderem Prozess): `<source> ? <target variable>`

Output (schreiben an anderen Prozess): `<destination> ! <expression>`

Ich schreie dich an

→ Synchronization via Communication → Rendezvous zwischen involvierten Prozessen

Syntax (2)

Parallel Command

```
<process> || <process> …
```

Concurrent Execution der Prozesse

“parallel command terminates successfully only if and when they have all successfully terminated”

Syntax (3)

Loops und Bedingungen

```
        *[ <guard> -> <command list> [] <guard> -> <command list> … ]
        *[ x >= y  -> m := x          [] y >= x  -> m := y ]
i = 0; *[ i < size, content(i) != n -> i := i+1 ]

        *[ (range)   <guard>          -> <command list>]
        *[ (i:1..10) console(i) ? C -> … ]
```

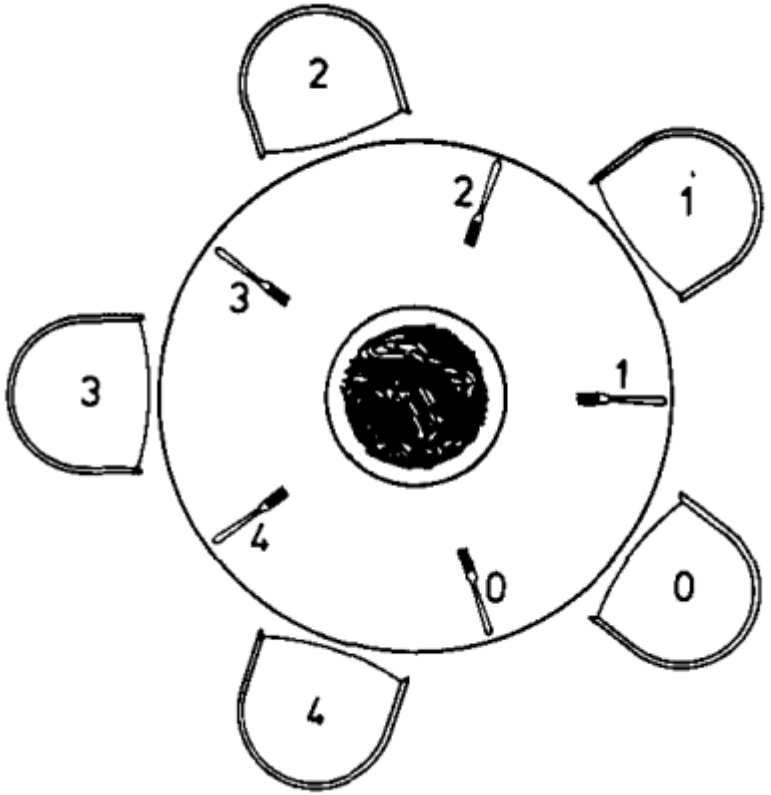
Zuerst wird der `<guard>` ausgeführt und nur wenn dieser erfolgreich war die `<command list>`

Dining Philosophers

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

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

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( )
          ]
```



Das Wesentliche

- Prozesse sind **unabhängig** voneinander (kein Shared Memory)
- Prozesse kommunizieren über «**Message Passing**»
- Kommunikation passiert **Synchron**

Und danach? (1)


```
PAR
  SEQ
    EKanal1 ? a
    EKanal2 ? b
    c := a * b
    AKanal1 ! c
  SEQ
    EKanal3 ? x
    EKanal4 ? y
    z := x + y
    AKanal2 ! z
```

```
ALT
  count1 < 100 & c1 ? data
    SEQ
      count1 := count1 + 1
      merged ! data
  count2 < 100 & c2 ? data
    SEQ
      count2 := count2 + 1
      merged ! data
  status ? request
    SEQ
      out ! count1
      out ! count2
```

A Tour of Go

Channels are a typed conduit through which you can send and receive values with the channel operator, `<-`.

```
ch <- v    // Send v to channel ch.
v := <-ch  // Receive from ch, and
           // assign value to v.
```

(The data flows in the direction of the arrow.)

Like maps and slices, channels must be created before use:

```
ch := make(chan int)
```

By default, sends and receives block until the other side is ready. This allows goroutines to synchronize without explicit locks or condition variables.

Und danach? (2)

```
for {
    select {
    case c <- x:
        x, y = y, x+y
    case <-quit:
        fmt.Println("quit")
        return
    }
}
```