Evolution of Software Languages

Theo D'Hondt

Bachelor of Computer Science
Faculty of Sciences and Bio-Engineering Sciences
Vrije Universiteit Brussel
Academic Year 2015-2016
Evolution of Software Languages

Tony Hoare: CSP

https://en.wikipedia.org/wiki/Tony_Hoare

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, parallel programs, program structure, parallel program composition, concurrency, input, output, guarded commands, nondeterminacy, counterparts, procedures, multiple entries, multiple exits, classes, data representations, recursion, and concurrent critical regions, monitors, iterative arrays

C. A. R. Hoare

1. Introduction

Among the primitive concepts of computer programming, and of the high level languages in which programs are expressed, the notion 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, those basic constructs have received widespread recognition and use. A repetitive construct (e.g. the while loop), an alternative construct (e.g. the conditional if-them-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: Subprograms (Procedures), procedures (Algo 604[1]) were written (PL/I), commands (Fortran 77), classes (COBOL 67[3]), processes and monitors (Concurrent Pascal [2]), thieves (CCL [1]), forms (ALGOL 68 [3]), 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, capable, reliable, and economical than a machine which is disguised as a monoproccessor.

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 common stores (as in Algol 68 [4], 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. semaphore switches) and unsuitability (e.g. glitches) in some technologies of hardware implementation. A greater variety of methods has been proposed for synchronization: semaphores [5], 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 simple solution to all these problems. The essential proposals are:

1. Dijkstra's guarded commands [9] 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.

Simple forms of input and output command are introduced. They are used for communication between concurrent processes.
Dining philosophers

FORK

* [ phil(i)?pickup() → phil(i)?putdown()
    []
    phil(i+1 mod 5)?pickup() → phil(i+1 mod 5)?putdown() ]

loop command
Dining philosophers

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

**loop command**

- loop body forever
Dining philosophers

FORK
* [ phil(i)?pickup() → phil(i)?putdown()
[ phil((i+1) mod 5)?pickup() → phil((i+1) mod 5)?putdown() ]

guarded command
Dining philosophers

FORK

* [ phil(i)?pickup() → phil(i)?putdown()
  []
  phil((i+1) mod 5)?pickup() → phil((i+1) mod 5)?putdown() ]

guarded command

• if no guard open: do nothing
• if one guard open: perform guard and corresponding body
• if multiple guards open: select one at random
Dining philosophers

FORK

* [phil(i)?pickup() → phil(i)?putdown()]
  phil(i+1 mod 5)?pickup() → phil(i+1 mod 5)?putdown()
]

input guards

an input guard p?q(x,y,...) is open if p is waiting to output a pattern q(a,b,...)
Dining philosophers

FORK
   *
      [ phil(i)?pickup() \rightarrow phil(i)?putdown() ]
      [ phil(i+1 mod 5)?pickup() \rightarrow phil(i+1 mod 5)?putdown() ]

input commands
Dining philosophers

FORK
  *
 [phil(i)?pickup() → phil(i)?putdown()
  []
  phil(i+1 mod 5)?pickup() → phil(i+1 mod 5)?putdown()
]

input commands

for p?q(x,y,...)

• wait for process p to wait to output a pattern q(a,b,...)
• bind assign a to x, b to y, ...
• proceed with p and current process
Dining philosophers

FORK
  * [  
    phil(i)?pickup() → phil(i)?putdown()
    []
    phil(i+1 mod 5)?pickup() → phil(i+1 mod 5)?putdown()
  ]

PHIL
  * [  
    THINK;
    room!enter();
    fork(i)!pickup();
    fork(i+4 mod 5)!pickup();
    EAT;
    fork(i+4 mod 5)!putdown();
    fork(i)!putdown();
    room!exit()
  ]
Dining philosophers

FORK
  * [phil(i)?pickup() → phil(i)?putdown()
    []
    phil(i+1 mod 5)?pickup() → phil(i+1 mod 5)?putdown()]

PHIL
  * [
    THINK:
    room!enter();
    fork(i)!pickup();
    fork(i+4 mod 5)!pickup();
    EAT:
    fork(i+4 mod 5)!putdown();
    fork(i)!putdown();
    room!exit();
  ]

output commands for p!q(exp₁,exp₂,...)
• evaluate a = exp₁, b = exp₂, ...
• wait for process p to request input to a pattern q(x,y,...)
• bind assign a to x, b to y, ...
• proceed with p and current process
Dining philosophers

\[
\text{FORK} \\
\land\quad \begin{array}{l}
\text{phil}(i)\text{?pickup()} \rightarrow \text{phil}(i)\text{?putdown()}
\
\text{phil}(i+1 \text{ mod } 5)\text{?pickup()} \rightarrow \text{phil}(i+1 \text{ mod } 5)\text{?putdown()}
\end{array}
\]

\[
\text{PHIL} \\
\land\quad \begin{array}{l}
\text{THINK};
\
\text{room}!\text{enter()};
\
\text{fork}(i)!\text{pickup()};
\
\text{fork}(i+4 \text{ mod } 5)!\text{pickup()};
\
\text{EAT};
\
\text{fork}(i+4 \text{ mod } 5)!\text{putdown()};
\
\text{fork}(i)!\text{putdown()};
\
\text{room}!\text{exit()}
\end{array}
\]

\[
\text{ROOM} \\
\land\quad \begin{array}{l}
\text{count}:\text{integer};
\
\text{count} := 0;
\
\land\quad \begin{array}{l}
(i:0,4):\text{count}<4 \& \text{phil}(i)\text{?enter()} \rightarrow \text{count} := \text{count}+1
\
(i:0,4):\text{phil}(i)\text{?exit()} \rightarrow \text{count} := \text{count}-1
\end{array}
\end{array}
\]
Dining philosophers

FORK
  * [ phil(i)?pickup() \rightarrow phil(i)?putdown() ]

PHIL
  * [ THINK;
      room!enter();
      fork(i)!pickup();
      fork(i+4 mod 5)!pickup();
      EAT;
      fork(i+4 mod 5)!putdown();
      fork(i)!putdown();
      room!exit() ]

ROOM
  [ count:integer;
    count := 0;
    * [ (i:0,4):count<4 & phil(i)?enter() \rightarrow count := count+1 ]
    [ (i:0,4):phil(i)?exit() \rightarrow count := count-1 ]
  ]

local variables
Dining philosophers

FORK

* [ phil(i)?pickup() → phil(i)?putdown()
   [] phil(i+1 mod 5)?pickup() → phil(i+1 mod 5)?putdown() ]

PHIL

* [
   THINK;
   room!enter();
   fork(i)!pickup();
   fork(i+4 mod 5)!pickup();
   EAT;
   fork(i+4 mod 5)!putdown();
   fork(i)!putdown();
   room!exit()
]

ROOM

[ count:integer;
  count := 0;
  *
   (i:0,4):count<4 & phil(i)?enter() → count := count+1
   []
   (i:0,4):phil(i)?exit() → count := count-1
]

local variables

local & lexical scope
Dining philosophers

FORK
*  
   phil(i)?pickup() → phil(i)?putdown()
   []
   phil((i+1 mod 5)?pickup() → phil((i+1 mod 5)?putdown()
   ]

PHIL
*  
   THINK;
   room!enter();
   fork(i)!pickup();
   fork((i+4 mod 5)!pickup();
   EAT;
   fork((i+4 mod 5)!putdown();
   fork(i)!putdown();
   room!exit()
   ]

ROOM
  [  
   count:integer;
   count := 0;
   *  
      (i:0,4):count<4 & phil(i)?enter() → count := count+1
      []
      (i:0,4):phil(i)?exit() → count := count-1
   ]

boolean guard
Dining philosophers

FORK
    *
    [  
        phil(i)?pickup() → phil(i)?putdown()  
        []  
        phil(i+1 mod 5)?pickup() → phil(i+1 mod 5)?putdown()  
    ]

PHIL
    *
    [  
        THINK;  
        room!enter();  
        fork(i)!pickup();  
        fork(i+4 mod 5)!pickup();  
        EAT;  
        fork(i+4 mod 5)!putdown();  
        fork(i)!putdown();  
        room!exit()  
    ]

ROOM
    [  
        count:integer;  
        count := 0;  
        *
        [  
            (i:0,4):count<4 & phil(i)?enter() → count := count+1  
            []  
            (i:0,4) phil(i)?exit() → count := count-1  
        ]  
    ]

iterators
Dining philosophers

FORK
  * [ phil(i)?pickup() → phil(i)?putdown() [] phil(i+1 mod 5)?pickup() → phil(i+1 mod 5)?putdown() ]

PHIL
  * [ THINK; room!enter(); fork(i)!pickup(); fork(i+4 mod 5)!pickup(); EAT; fork(i+4 mod 5)!putdown(); fork(i)!putdown(); room!exit() ]

ROOM
  [ count:integer; count := 0; * [ (i:0,4):count<4 & phil(i)?enter() → count := count+1 [] (i:0,4):phil(i)?exit() → count := count-1 ] ]

MAIN
  [ fork(i:0,4)::FORK || phil(i:0,4)::PHIL || room:ROOM ]
Dining philosophers

FORK
  * [ phil(i):pickup() → phil(i):putdown() ]
  phil(i+1 mod 5):pickup() → phil(i+1 mod 5):putdown() ]

PHIL
  * [
    THINK;
    room!enter();
    fork(i):pickup();
    fork((i+4 mod 5)):pickup();
    EAT;
    fork((i+4 mod 5)):putdown();
    fork(i):putdown();
    room!exit() ]

ROOM
 [ count:integer;
   count := 0;
 ]
  * [
    (i:0,4):count<4 & phil(i):enter() → count := count+1
    (i:0,4):phil(i):exit() → count := count-1 ]
]

MAIN
  [ fork((i:0,4)::FORK || phil((i:0,4)::PHIL || room::ROOM ]
Dining philosophers

FORK

* [ phil(i)?pickup() → phil(i)?putdown() [] phil(i+1 mod 5)?pickup() → phil(i+1 mod 5)?putdown() ]

PHIL

* [ THINK; room!enter(); fork(i)!pickup(); fork(i+4 mod 5)!pickup(); EAT; fork(i+4 mod 5)!putdown(); fork(i)!putdown(); room!exit() ]

ROOM

[ count:integer; count := 0; * [ (i:0,4):count<4 & phil(i)?enter() → count := count+1 [] (i:0,4):phil(i)?exit() → count := count-1 ] ]

MAIN

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

non-local scope is immutable!

parallel command
Dining philosophers (OCCAM)

CHAN PickUp[4], PutDown[4], Enter[4], Exit[4]:
Dining philosophers (OCCAM)

CHAN PickUp[4], PutDown[4], Enter[4], Exit[4]:

Using channels rather than explicitly naming peer process
Dining philosophers (OCCAM)

CHAN PickUp[4], PutDown[4], Enter[4], Exit[4]:

PROC Philosopher(VALUE identity) =
  WHILE TRUE
  SEQ
    Think
    Enter[identity]!ANY
    PickUp[identity]!ANY
    PickUp[identity+1 MOD 5]!ANY
    Eat
    PutDown[identity+1 MOD 5]!ANY
    PutDown[identity]!ANY
    Exit[identity]!ANY:
Dining philosophers (OCCAM)

CHAN PickUp[4], PutDown[4], Enter[4], Exit[4]:

PROC Philosopher(VALUE identity) =
WHILE TRUE
SEQ
Think
Enter[identity]!ANY
PickUp[identity]!ANY
PickUp[identity+1 MOD 5]!ANY
Eat
PutDown[identity+1 MOD 5]!ANY
PutDown[identity]!ANY
Exit[identity]!ANY:

PROC Forks =
VAR
Free[4]:
WHILE TRUE
ALT i = [0 FOR 4]
Free[i] & PickUp[i]?ANY
Free[i] := FALSE
PutDown[i]?ANY
Free[i] := TRUE:
Dining philosophers (OCCAM)

CHAN PickUp[4], PutDown[4], Enter[4], Exit[4]:

PROC Philosopher(VALUE identity) =
    WHILE TRUE
    SEQ
        Think
        Enter[identity]!ANY
        PickUp[identity]!ANY
        PickUp[identity+1 MOD 5]!ANY
        Eat
        PutDown[identity+1 MOD 5]!ANY
        PutDown[identity]!ANY
        Exit[identity]!ANY:

PROC Forks =
    VAR
        Free[4] :
    WHILE TRUE
        ALT i = [0 FOR 4]
            Free[i] & PickUp[i]!ANY
            Free[i] := FALSE
        PutDown[i]!ANY
            Free[i] := TRUE :

PROC Room =
    VAR
        Count :
    SEQ
        Count := 0
    WHILE TRUE
        ALT i = [0 FOR 4]
            Count<4 & Enter[i]!ANY
            Count := Count+1
        Exit[i]!ANY
            Count := Count-1 :
Dining philosophers (OCCAM)

CHAN PickUp[4], PutDown[4], Enter[4], Exit[4]:

PROC Philosopher(VALUE identity) =
  WHILE TRUE
  SEQ
    Think
    Enter[identity]!ANY
    PickUp[identity]!ANY
    PickUp[identity+1 MOD 5]!ANY
    Eat
    PutDown[identity+1 MOD 5]!ANY
    PutDown[identity]!ANY
    Exit[identity]!ANY:

PROC Forks =
  VAR
    Free[4] :
  WHILE TRUE
    ALT i = [0 FOR 4]
    Free[i] & PickUp[i]?ANY
    Free[i] := FALSE
    PutDown[i]?ANY
    Free[i] := TRUE:

PROC Room =
  VAR
    Count :
  SEQ
    Count := 0
  WHILE TRUE
    ALT i = [0 FOR 4]
    Count<4 & Enter[i]?ANY
    Count := Count+1
    Exit[i]?ANY
    Count := Count-1:

PAR
  Forks
  Room
  PAR i = [0 FOR 4]
  Philosopher(i)
Transputers

- System Services
- 2 Kbytes static RAM (50ns)
- Memory interface
- 32 bits processor
- Link
- Event
- Memory interface
- Link
- Link
- Link
### Timing characteristics

<table>
<thead>
<tr>
<th>class</th>
<th>operation</th>
<th>cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>arithmetic:</td>
<td>+, -</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>*</td>
<td>39</td>
</tr>
<tr>
<td></td>
<td>/</td>
<td>40</td>
</tr>
<tr>
<td>relational:</td>
<td>=, &gt;, &lt;, &lt;=, &gt;=</td>
<td>3</td>
</tr>
<tr>
<td>logical:</td>
<td>and</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>or</td>
<td>8</td>
</tr>
<tr>
<td>shift:</td>
<td>&gt;&gt;n, &lt;&lt;n</td>
<td>3+n</td>
</tr>
<tr>
<td>control:</td>
<td>seq</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>if {n alt}</td>
<td>1.4+4.3n</td>
</tr>
<tr>
<td></td>
<td>while</td>
<td>12</td>
</tr>
<tr>
<td>assignment:</td>
<td>:=</td>
<td>1.1</td>
</tr>
</tbody>
</table>
Channel commands

<channel> ? <data>

receive <data> from <channel>

<channel> ! <data>

send <data> through <channel>
Hypercube

000 100 110
001 101 111
010 011

dimension 001
dimension 010
dimension 100

Evolution of Software Languages

Computational Aspects of VLSI
Jeffrey D. Ullman
Computer Science Press 1984
Regular grid
Cube-connected cycles
Square grid
Database controller

![Diagram of a database controller system with components such as System Services, 1Kbyte static RAM, Disk interface, Memory interface, 16 bits processor, 4Kbytes ROM, Link, and Event.]
Database controller
Development_2

scsi-to-link adapter
3-to-1 multiplexer

BYTE x:
CHAN output:
[3]CHAN input:
ALT i=0 FOR 2
input[i]?x
output!x
Sort pipeline

\[ \text{c[0]} = \text{producer} \]
\[ \text{PAR } i=0 \text{ for } n-1 \]
\[ \text{INT } x, y: \]
\[ \text{SEQ} \]
\[ \text{c[i]}?x \]
\[ \text{SEQ } j=1 \text{ FOR } n-i \]
\[ \text{c[i]}?y \]
\[ \text{IF} \]
\[ x > y \]
\[ \text{c[i+1]}!y \]
\[ x \leq y \]
\[ \text{SEQ} \]
\[ \text{c[i+1]}!x \]
\[ x := y \]
\[ \text{c[i+1]}!x \]
\[ \text{SEQ } j=1 \text{ FOR } i \]
\[ \text{c[i]}?x \]
\[ \text{c[i+1]}!x \]
Sort pipeline

[\text{n}] CHAN c:
PAR i=0 for \text{n-1}
INT x, y:
SEQ
\text{c[i]}?x
SEQ j=1 FOR \text{n-i}
\text{c[i]}?y
IF
x > y
\text{c[i+1]}!y
x \leq y
SEQ
\text{c[i+1]}!x
x := y
\text{c[i+1]}!x
SEQ j=1 FOR i
\text{c[i]}?x
\text{c[i+1]}!x
Ray-tracing

high-performance graphics system
Conclusion

• Static processes
• No asynchronous communication
• Poor data/procedural constructs
• Cfr. ADA
Assignment

1. Read the 1978 CSP paper
2. Check XMOS