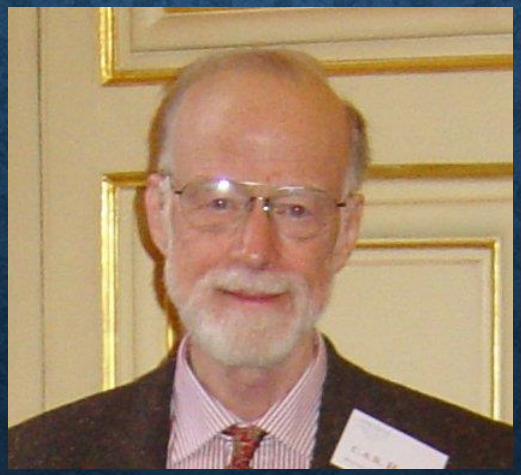
COMMUNICATING SEQUENTIAL PROCESSES CSP

C.A.R. Hoare (1978)

Communications of the ACM, Vol. 21, No. 8, pp. 666-677.

Tony Hoare

1980 Turing award - For his fundamental contributions to the definition and design of programming languages.



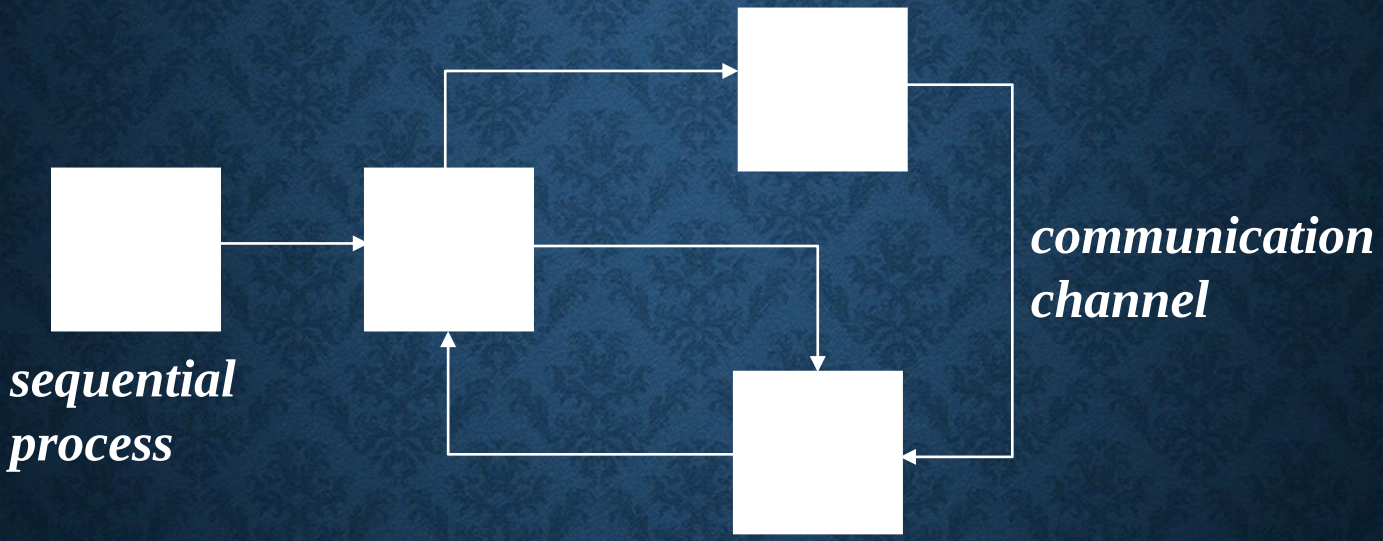
Agenda

Introduction

Commands

Examples

Parallel composition of CSP is a fundamental program structuring method

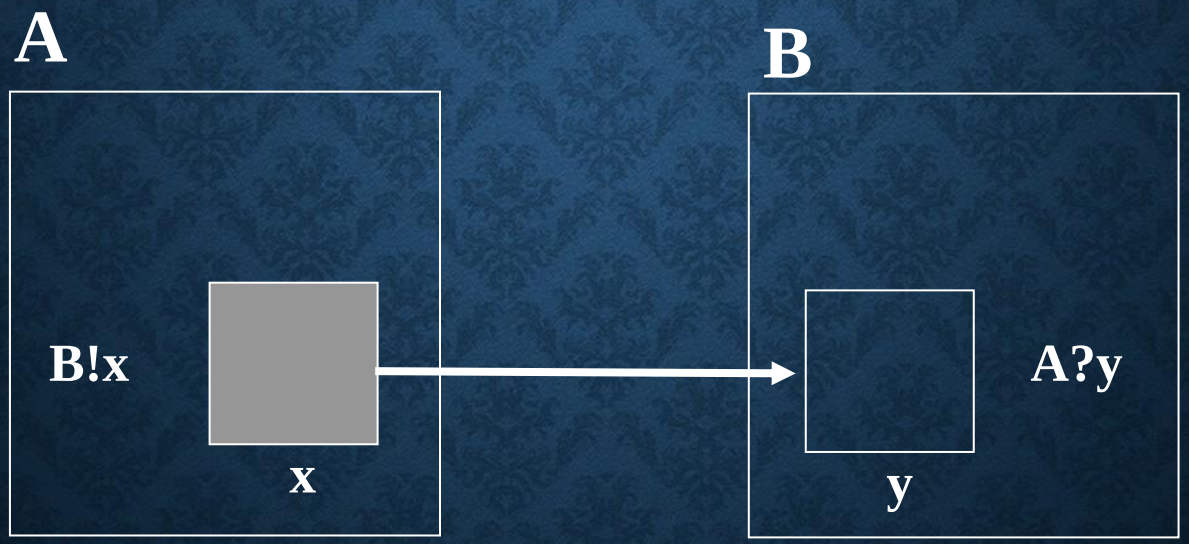


- single thread of control
- autonomous
- encapsulated
- named
- static

- synchronous
- reliable
- unidirectional
- point to point
- fixed topology

Input and output are basic primitives of programming

operators: ! (send)
? (receive)



A CSP program consists of a sequence of commands

The symbols [and] are respectively analogous to begin and end in Algol.

The semicolons are used to indicate sequential execution.

There are four 'standard' types available: real, integer, boolean, and character.

An array is declared as follows: p:(1 .. 100) integer; where p is a one dimensional array of type integer having 100 elements.

A CSP program consists of a sequence of commands

Commands can either succeed of fail

For example, a familiar assignment command $a := b$
will fail if either

- b is undefined, or
- the types of a and b do not match

Agenda

Introduction

Commands

Examples

A parallel command specifies concurrent execution of its constituent processes

<parallel command> ::= <*command list1*> || <*command list2*>

<CL1> || <CL2> || <...> || <CLn>

can have more than two processes

producer:: <CL1> || consumer:: <CL2>

can add process name (label)

I/O commands specify communication between two sequential processes

<input command >::= <source process> ? <target variable>

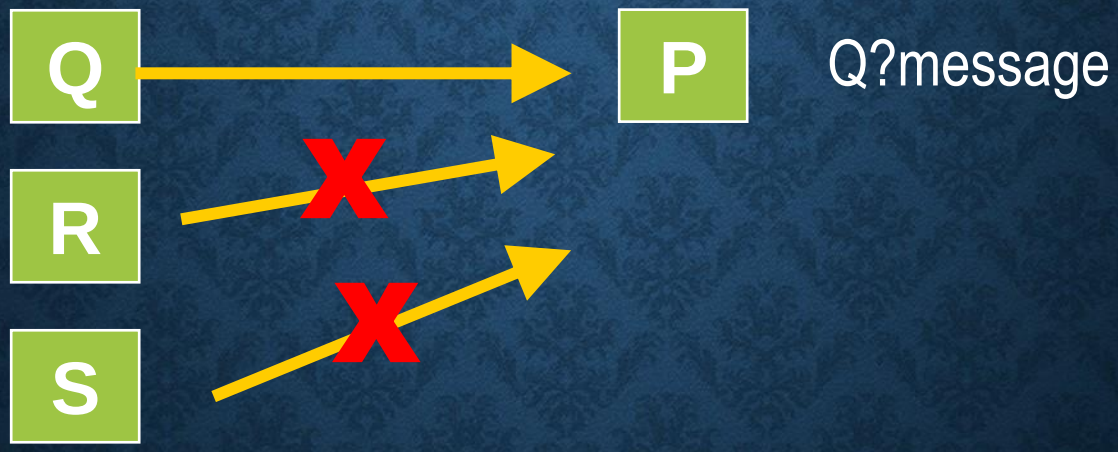
keyboard ? m

<output command >::= <destination process> ! <expression>

screen ! average

Non-buffered (synchronous) message passing with
explicit naming of source and destination processes

Blocking receive with *direct naming* allows P to receive a message from Q



Guarded and alternative commands let processes wait for messages coming from different sources

A guarded command is executed only if and when the execution of its guard does not fail

<guarded command> ::= <guard> → <command list>

in<out + 10; producer?buffer(in mod 10) →

A guard consists of a list of boolean expressions

An alternative command specifies execution of one of its constituent guarded commands

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

(□ <guarded command>)]

[$x \geq y \rightarrow max := x$

□ $y \geq x \rightarrow max := y$]

Arbitrary selection

Process P can now wait for first message from any of three senders Q, R and S

[

Q ? msg → <process msg from Q> □

R ? msg → <process msg from R> □

S ? msg → <process msg from S>

]

An input command is false if there is no message pending

A repetitive command specifies as many iterations as possible

<repetitive command> ::= *<alternative command>

* [i > 0 → fact := fact*i ; i := i–1]

Repetitive, alternative and multiple guarded commands

The **repetitive command** (the ∗ operator) over the **alternative command** (the □ operator) on **multiple guarded commands** (each having the form Gi → CLi) is used as follows:

* [G1 → CL1 □ G2 → CL2 □ … □ Gk → CLk]

[(i:1..k) G → CL]

A process whose label subscripts include ranges stands for a series of processes

Agenda

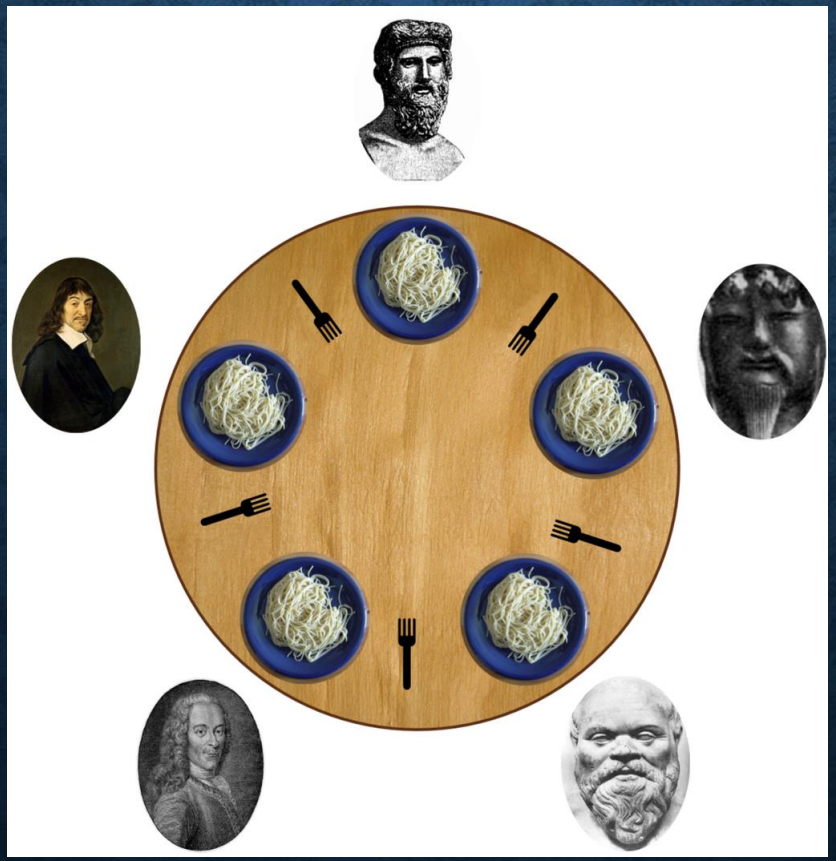
Introduction

Commands

Examples

Five PHILosophers share a circular table

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



https://en.wikipedia.org/wiki/Dining_philosophers_problem#/media/File:An_illustration_of_the_dining_philosophers_problem.png

The dining philosopher *i* spends his live thinking and eating

```
PHIL =
 *[... during i-th 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 fork *i* is picked up and 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
  ]
```

A PRODUCER sends data to a CONSUMER using a bounded buffer X

[producer::PRODUCER || X || consumer::CONSUMER]

```
PRODUCER =
data:real;
*[... during its lifetime ... →
        WORK;
        X!data
  ]
```

```
CONSUMER =
data:real;
*[... during its lifetime ... →
        X!more( );
        X?data;
        WORK
  ]
```

The bounded buffer smooths variations in the speed of producer and consumer processes

```
X::
buffer: (0..9) real;
in, out : integer; in:=0; out:=0;
 *[
        in<out + 10; producer?buffer(in mod 10)  →
                                        in:=in + 1
          □
        out < in; consumer?more() →
                                        consumer!buffer(out mod 10);
                                        out:=out + 1
 ]
```

An array of client processes access a single shared resource protected by semaphore S

```
[  X(i:1..100)::CP || S ]

CP =
 *[... during its lifetime ... →
            WORK;
            S!P( );
            CRITICAL SECTION;
            S!V( )
  ]
```

The semaphore is used by the array of client processes

```
S::
val:integer; val:= 1;
   *[
          (i:1..100)  val>0; X(i)?P() → val := val - 1
          □
          (i:1..100) X(i)?V() → val := val + 1
     ]
```

Input, output and concurrency should be regarded as primitives of programming