Communicating Sequential Processes

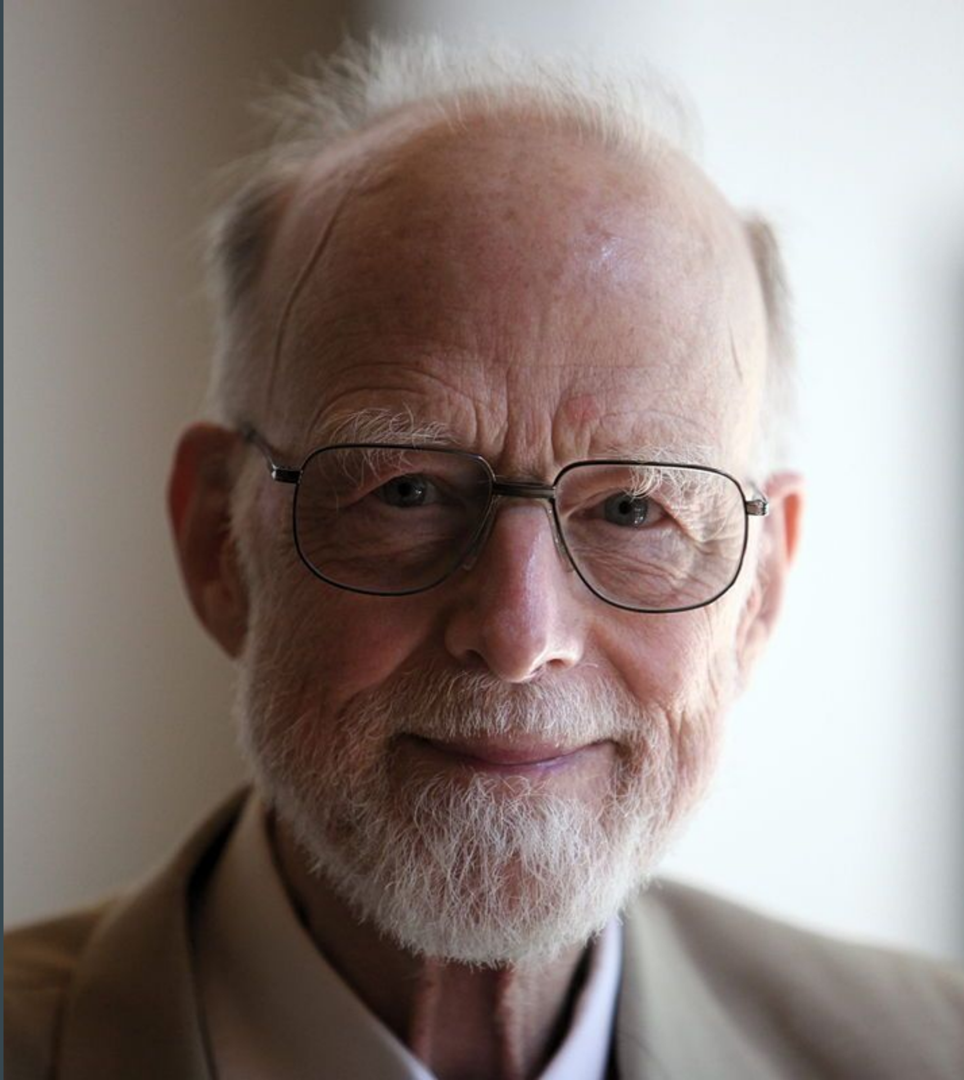
overview of a 1978 paper by C.A.R. Hoare

Sir Tony Hoare

Hoare logic

Quicksort

CSP



1. Introduction

- Assignment:
  - well understood
  - built into programming languages

1. Introduction

- Assignment:
  - well understood
  - built into programming languages

- Input/output:
  - *not* well understood
  - often left to libraries

1. Introduction

- Programming languages usually agree on having:
  - repetitive constructs (loops)
  - alternative constructs (conditionals)
  - sequential composition

1. Introduction

- Programming languages usually agree on having:
  - repetitive constructs (loops)
  - alternative constructs (conditionals)
  - sequential composition
- but *not* on the design of **abstractions**:
  - subroutines (Fortran)
  - coroutines (Unix)
  - classes (Simula 67)
  - actors (*Actor model* by *Carl Hewitt, 1973*)

Traditional computer was designed for

“deterministic execution of a single sequential program”

CDC 6600

Used at CERN since 1965

Had 10 parallel functional units

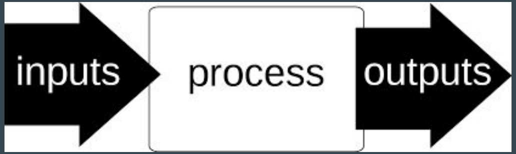
Multiprocessor machine

"… may become more powerful, capatious, reliable, and economical than a machine which is disguised as a monoprocessor."

Need for synchronization

- Shared mutable state
- Semaphores
- Monitors
- Queues

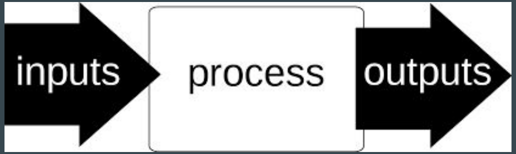
CSP's Conjecture



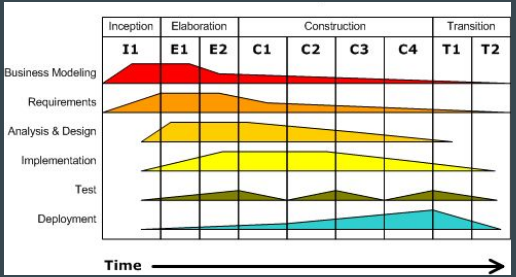
“Input and output

are basic primitives of programming“

CSP's Conjecture



“Input and output
are basic primitives of programming“



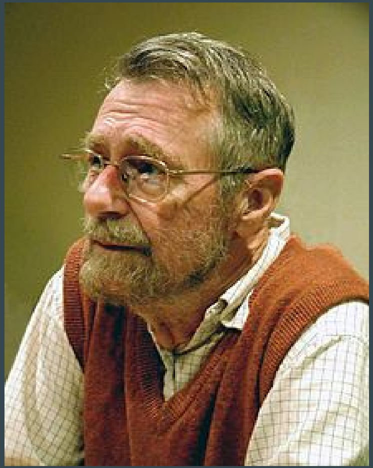
“Parallel composition of
communicating sequential processes
is a fundamental program structuring method“

CSP's Conjecture

- These concepts are “surprisingly versatile” when combined with Dijstra’s guarded commands

Guarded Command Language

Edsger Dijkstra



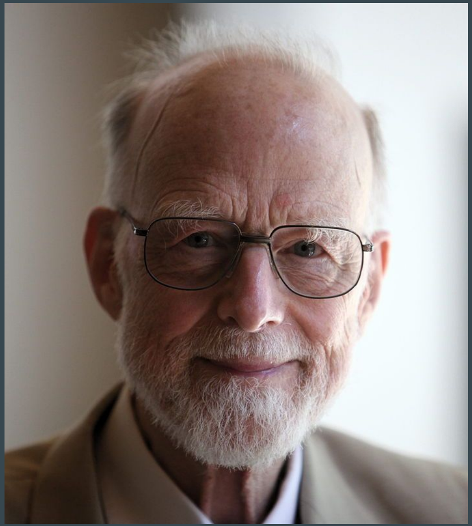
A simple pseudo-language that makes it easier to prove the correctness of programs.

```
    Guards      Statements
if a >= b  →  max := a
 | b >= a  →  max := b
fi
```

```
    Guards      Statements
do a > b  →  a := a - b
 | b > a  →  b := b - a
od
```

CSP Language

Based on Guarded Command Language



by Tony Hoare

2. Concepts and Notations

- Assignment Command

(1) $x := x + 1$

(2) $(x, y) := (y, x)$

(3) $x :=$ cons(left, right)

(4) cons(left, right) $:= x$

(5) insert(n) $:=$ insert($2 * x + 1$)

(6) $c :=$ P( )

2. Concepts and Notations

- Input Command
- Output Command

(1) cardreader?cardimage

(2) lineprinter!lineimage

(3) $X?(x, y)$

(4) DIV!$(3*a + b, 13)$

2. Concepts and Notations

- Alternative Command

MAX $[x \geq y \rightarrow m := x \,[]\, y \geq x \rightarrow m := y]$

- Repetitive Command

SEARCH $i := 0; *[i < \text{size}; \text{content}(i) \neq n \rightarrow i := i + 1]$

2. Concepts and Notations

- Parallel Command

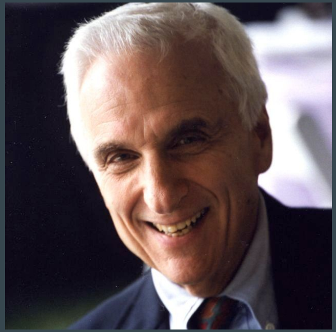
[cardreader?cardimage||lineprinter!lineimage]

$X(i{:}1..n) \;::\; \mathrm{CL}$

❖ "mutually disjoint"

3. Coroutines

First coined by *Melvin Conway* and published in his 1963 paper:

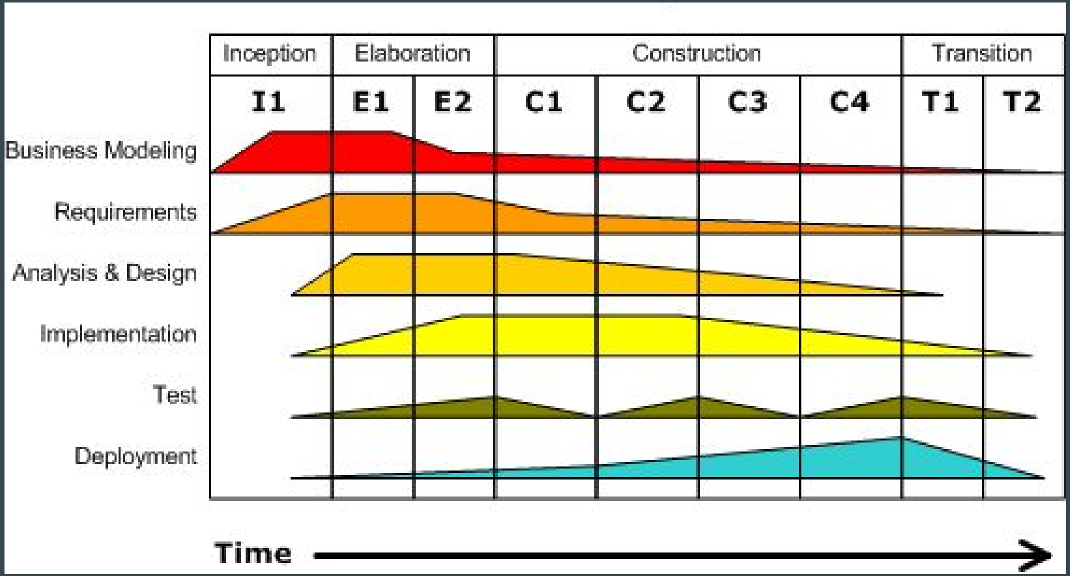


Coroutine ... an autonomous program which communicates with adjacent modules as if they were *input* or *output* subroutines.

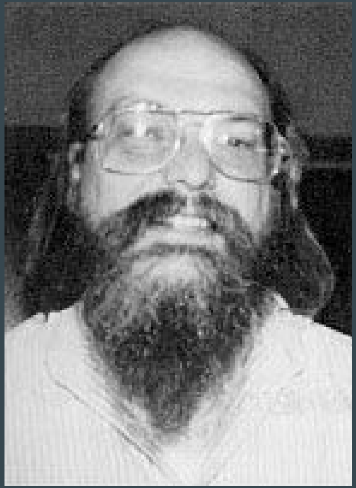
Thus, coroutines are subroutines *all at the same level*, each acting as if it were the master program when in fact *there is no master program*.

from "Design of a Separable Transition-Diagram Compiler"

3. Coroutines



3. Coroutines



3. Coroutines

COPY

```
X :: *[c:character; west?c → east!c]
```

SQUASH

```
X :: *[c:character; west?c →
   [c ≠ asterisk → east!c
   []c = asterisk → west?c;
         [c ≠ asterisk → east!asterisk; east!c
         []c = asterisk → east!upward arrow
   ] ]   ]
```

3. Coroutines

DISASSEMBLE

```
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
]
```

ASSEMBLE

```
*[cardimage:(1..80)character; cardfile?cardimage →
        i:integer; i := 1;
        *[i ≤ 80 → X!cardimage(i); i := i + 1]
        X!space
]
```

3. Coroutines (parallel composition)

REFORMAT

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

CONWAYS_PROBLEM

[west::DISASSEMBLE||X::SQUASH||east::ASSEMBLE]

4. Subroutines and Data Representations

Coroutines can be used to simulate:

- subroutines
- recursion
- ADTs / objects / actors

4. Subroutines and Data Representations

Subroutine

```
[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. Subroutines and Data Representations

Recursion

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

4. Subroutines and Data Representations

ADT

SET

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

5. Monitors and Scheduling

Monitor

$$*[(i{:}1..100)X(i)?(\text{value parameters}) \rightarrow \ldots;\ X(i)!(\text{results})]$$

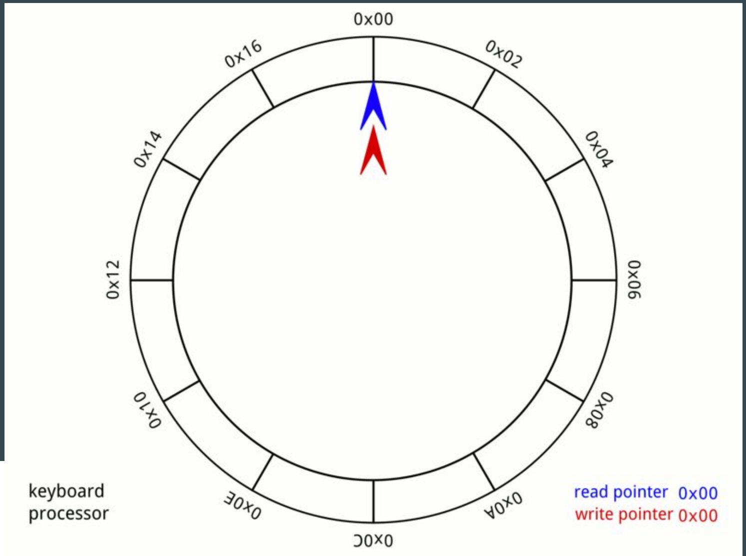
5. Monitors and Scheduling

Semaphore

```
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
    ]
```

5. Monitors and Scheduling

Bounded Buffer




```
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
  ]
```

5. Monitors and Scheduling

Dining Philosophers

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

5. Monitors and Scheduling

Dining Philosophers

```
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( )
           ]
```

5. Monitors and Scheduling

Dining Philosophers

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

5. Monitors and Scheduling

Dining Philosophers

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

6. Miscellaneous

7. Discussion

- APL-like brevity *(7.1)*
- Input/output notation is not assignment *(7.1)*

7. Discussion

- Explicit source/destination makes processes hard to reuse (e.g. in libraries) *(7.2)*

7. Discussion

- *Channels (7.3)* to connect ports exposed by processes
  - *option*: multichannels

7. Discussion

- *Buffers (7.4)* to hold output messages
  - *option*: automatic buffering (actor-like “mailboxes”)

7. Discussion

- Possibly introduce unbounded process arrays *(7.5)*

7. Discussion

- *Fairness (7.6)* is hard to ensure
  - should be the programmer's responsibility
  - e.g. Go lang's `select` is unfair

```
*[continue; X?stop( ) → continue := false
  []continue → n := n + 1
  ]
```

7. Discussion

- CSP’s imperative approach is more machine-oriented *(7.7)*
  - vs. Gilles Kahn’s “Language for parallel programming” which is strictly deterministic (e.g. no alternative commands)

7. Discussion

- *Output guards (7.8)* is an option
  - e.g. Go lang's `select`
  - e.g. Clojure's `alt!`

$$Z :: [\mathrm{true} \rightarrow X!2;\ Y!3 \,[]\, \mathrm{true} \rightarrow Y!3;\ X!2]$$

7. Discussion

- Automatic termination of a repetitive command on termination of the sources of all its input guards *(7.9)* is
    - convenient VS. implementable?

8. Conclusion

- CSP is not a practical language
- CSP lacks more useful abstractions
- but CSP primitives can be used to construct:
  - objects
  - actors
  - monitors
  - streams
  - etc.

Questions

@ogrigas

github.com/ogrigas