paper-reading

理解顺序进程间通讯

Understanding Communicating Sequential Processes

Go 夜读 SIG 小组 | 欧长坤

第 69 期
Nov. 07, 2019

paper-reading

理解顺序进程过程间通讯

Understanding Communicating Sequential Processes

Go 夜读 SIG 小组 | 欧长坤

第 69 期
Nov. 07, 2019

主要内容

CSP 理论的诞生背景

论文的主要内容与结论

反思与总结

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.

为什么要读这篇论文？

1. 并行计算领域的传世之作
2. 为理解后续 CSP 理论铺路
3. 通过实现论文养成一种利用通信来共享内存的思维

…

产生背景

History

程序的常见结构

传统程序设计中三种常见的构造结构：重复构造（for）、条件构造（if）、顺序组合（;）

除此之外，其他的一些结构：

- Fortran: Subroutine
- Algol 60: Procedure
- PL/I: Entry
- UNIX: Coroutine
- SIMULA 64: Class
- Concurrent Pascal: Process and monitor
- CLU: Cluster
- ALPHARD: Form
- Hewitt: Actor

程序的演进史：https://spf13.com/presentation/the-legacy-of-go/

并行的兴起

并行性的引入：

- 硬件：CDC 6600 并行单元

- 软件：I/O 控制包、多编程操作系统

处理器技术在多核处理器上提出一组自洽的处理器可以更加高效、可靠
但为了使用机器的效能，必须在处理器之间进行通信与同步。为此也提出了很多方法

- 公共存储的检查与更新：Algol 68，PL/I，各种不同的机器码（开销较大）

- semaphore

- events: PL/I

- 条件临界区

- monitors and queues: Concurrent Pascal

- path expression

这么多方法并没有一个统一的选择标准。

一种新的设计

简洁设计、解决所有问题

- Dijkstra's guarded command
  + 条件分支
  + 主要区别：若多个条件为真，则随机选择一个分支执行

- Dijkstra's parbegin parallel command
  + 启动并发过程

- input and output command
  + 过程间通信手段
  + 通信值存在复制
  + 没有缓存
  + 指令推迟到其他过程能够处理该消息

- input + guarded command
  + 等价于 Go 的 select 语句，条件分支仅当 input 源 ready 时开始执行
  + 多个 input guards 同时就绪，只随机选择一个执行，其他 guards 没有影响

- repetitive command
  + 循环
  + 终止性取决于其所有 guards 是否已终止

- pattern-matching

设计细节

Designs

概念与记号:程序结构

```
<cmd>            ::= <simple cmd> | <structured cmd>
<simple cmd>     ::= <assignment cmd> | <input cmd> | <output cmd>
<structured cmd> ::= <alternative cmd> | <repetitive cmd> | <parallel cmd>
<null cmd>       ::= skip
<cmd list>       ::= {<declaration>; | <cmd>; } <cmd>
```

概念与记号:并行指令

```
<parallel cmd>     ::= [<proc>{||<proc>}]
<proc>             ::= <proc label> <cmd list>
<proc label>       ::= <empty> | <identifier> :: | <identifier>(<label subscript>{,<label subscript>}) ::
<label subscript>  ::= <integer const> | <range>
<integer constant> ::= <numeral> | <bound var>
<bound var>        ::= <identifier>
<range>            ::= <bound variable>:<lower bound>..<upper bound>
<lower bound>      ::= <integer const>
<upper bound>      ::= <integer const>
```

举例：

```
X      (i      :  1      ..  n)     ::     CL
↑       ↑          ↑          ↑             ↑
identifier  bound var  lower bound  upper bound  cmd list
```

⇒

```
X(1) :: CL1 || X(2) :: CL2 || … || X(n) :: CLn
```

概念与记号:并行指令

```
<parallel cmd>     ::= [<proc>{||<proc>}]
<proc>             ::= <proc label> <cmd list>
<proc label>       ::= <empty> | <identifier> :: | <identifier>(<label subscript>{,<label subscript>}) ::
<label subscript>  ::= <integer const> | <range>
<integer constant> ::= <numeral> | <bound var>
<bound var>        ::= <identifier>
<range>            ::= <bound variable>:<lower bound>..<upper bound>
<lower bound>      ::= <integer const>
<upper bound>      ::= <integer const>
```

更多举例：

```
(1) [cardreader ? cardimage || lineprinter ! lineimage]

    // 两个并行过程

(2) [west::DISASSEMBLE || X::SQUASH || east::ASSEMBLE]

    // 三个并行过程

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

    // 十一个个并行过程，其中第二个和第三个并行过程分别包含五个并行的子过程
```

概念与记号:赋值指令

```
<assignment cmd>    ::= <target var> := <expr>
<expr>              ::= <simple expr> | <structured expr>
<structured expr>   ::= <constructor> ( <expr list> )
<constructor>       ::= <identifier> | <empty>
<expr list>         ::= <empty> | <expr> {, <expr> }
<target var>        ::= <simple var> | <structured target>
<structured target> ::= <constructor> ( <target var list> )
<target var list>   ::= <empty> | <target var> {, <target var> }
```

举例：

```
(1) x := x + 1                  // 普通的赋值
(2) (x, y) := (y, x)            // 普通的赋值
(3) x := cons(left, right)      // 结构体赋值
(4) cons(left, right) := x      // 如果 x 是一个结构体 cons(y, z)，则赋值成功 left:=y, right:=z，否则失败
(5) insert(n) := insert(2*x+1)  // 等价于 n := 2*x + 1
(6) c := P()                    // 空结构体，或称‘信号’
(7) P() := c                    // 如果 c 同为信号，则无实际效果，否则失败
(8) insert(n) := has(n)         // 不允许不匹配的结构体之间进行赋值
```

概念与记号:输入与输出指令

```
<input cmd>    ::= <source> ? <target var>
<output cmd>   ::= <destination> ! <expr>
<source>       ::= <proc name>
<destination>  ::= <proc name>
<proc name>    ::= <identifier> | <identifier> ( <subscripts> )
<subscripts>   ::= <integer expr> {, <integer expr> }
```

思考：根据例子（3）和（4），以下语句

[X :: DIV!(3*a+b, 13) || DIV :: X?(x,y)]

本质上是什么意思？

举例：

```
(1) cardreader?cardimage      // 类似于 cardimage <- cardreader
(2) lineprinter!lineimage     // 类似于 lineprinter := <- lineimage
(3) X?(x,y)                   // 从过程 X 接受两个值，并赋值给 x 和 y
(4) DIV!(3*a + b, 13)         // 向过程 DIV 发送两个值，分别为 3*a+b 和 13
(5) console(i)?c              // 向第 i 个 console 发送值 c
(6) console(j-1)!"A"          // 向第 j-1 个 console 发送字符 "A"
(7) X(i)?V()                  // 从第 i 个 X 接受一个信号 V
(8) sem!P()                   // 向 sem 发送一个信号 P
```

概念与记号:选择与重复指令

```
<repetitive cmd>   ::= * <alternative cmd>
<alternative cmd>  ::= [<guarded cmd> { □ <guarded cmd> }]
<guarded cmd>      ::= <guard> → <cmd list> | ( <range> {, <range> }) <guard> → <cmd list>
<guard>            ::= <guard list> | <guard list>;<input cmd> | <input cmd>
<guard list>       ::= <guard elem> {; <guard elem> }
<guard elem>       ::= <boolean expr> | <declaration>
```

举例：

```
(1) [x ≥ y → m := x □ y ≥ x → m := y]

    // 如果某个条件成立，则执行 → 后的语句；若均成立则随机选择一个执行

(2) i := 0; *[i < size; content(i) ≠ n → i := i+1]

    // 依次检查 content 中的值是否与 n 相同

(3) *[c:character; west?c → east!c]

    // 从 west 复制一个字符串并输出到 east 中
```

概念与记号：选择与重复指令

```
<repetitive cmd>   ::= * <alternative cmd>
<alternative cmd>  ::= [<guarded cmd> { □ <guarded cmd> }]
<guarded cmd>      ::= <guard> → <cmd list> | ( <range> {, <range> }) <guard> → <cmd list>
<guard>            ::= <guard list> | <guard list>;<input cmd> | <input cmd>
<guard list>       ::= <guard elem> {; <guard elem> }
<guard elem>       ::= <boolean expr> | <declaration>
```

更多举例：

```
(4) *[(i:1..10)continue(i); console(i)?c → X!(i,c); console(i)!ack(); continue(i) := (c ≠ sign off)]
    // 监听来自 10 个 console 的值，并将其发送给 X，当收到 sign off 时终止监听
(5) *[n:integer; X?insert(n) → INSERT □ n:integer; X?has(n) → SEARCH; X!(i<size)]
    // insert 操作：执行 INSERT
    // has 操作：执行 SEARCH，并当 i < size 时，向 X 发送 true，否则发送 false
(6) *[X?V() → val := val+1 □ val > 0; Y?P() → val := val-1]
    // V 操作：val++
    // P 操作：val > 0 时，val--，否则失败
```

算符

整个语言包含：

- 顺序算符 ;
- 并行算符 ||
- 赋值算符 :=
- 输入算符 ?（发送）
- 输出算符 !（接受）
- 选择算符 □
- 守卫算符 →
- 重复算符 *

协程

Coroutines

S31 COPY

编写一个过程 X，复制从 west 过程输出的字符到 east 过程

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

```
func S31_COPY(west, east chan rune) {
    for c := range west {
        east <- c
    }
    close(east)
}
```

S32 SQUASH

调整前面的程序，替换成对出现的「」为「↑」，假设最后一个字符不是「*」。**

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

练习：调整上面的程序，处理最后以奇数个「*」结尾的输入数据。

```
SQUASH_EX :: *[c:character; west?c →
  [ c != asterisk → east!c
   □ c = asterisk → west?c;
         [ c != asterisk → east!asterisk; east!c
          □ c = asterisk → east!upward arrow
+        ] □ east!asterisk
  ]    ]
```

```
func S32_SQUASH_EX(west, east chan rune) {
    for {
        c, ok := <-west
        if !ok {
            break
        }
        if c != '*' {
            east <- c
        }
        if c == '*' {
            c, ok = <-west
            if !ok {
+               east <- '*'
                break
            }
            if c != '*' {
                east <- '*'
                east <- c
            }
            if c == '*' {
                east <- '↑'
            }
        }
    }
    close(east)
}
```

S33 DISASSEMBLE

从卡片盒中读取卡片上的内容，并以流的形式将它们输出到过程 X ，并在每个卡片的最后插入一个空格。

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

```
func S33_DISASSEMBLE(cardfile chan []rune, X chan rune) {
    cardimage := make([]rune, 0, 80)
    for tmp := range cardfile {
        if len(tmp) > 80 {
            cardimage = append(cardimage, tmp[:80]...)
        } else {
            cardimage = append(cardimage, tmp[:len(tmp)]...)
        }
        for i := 0; i < len(cardimage); i++ {
            X <- cardimage[i]
        }
        X <- ' '
        cardimage = cardimage[:0]
    }
    close(X)
}
```

S34 ASSEMBLE

将一串流式字符串从过程 X 打印到每行 125 个字符的 lineprinter 上。必要时将最后一行用空格填满。

```
ASSEMBLE::
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
]
```

```
func S34_ASSEMBLE(X chan rune, lineprinter chan string)
{
    lineimage := make([]rune, 125)

    i := 0
    for c := range X {
        lineimage[i] = c
        if i <= 124 {
            i++
        }
        if i == 125 {
            lineimage[i-1] = c
            lineprinter <- string(lineimage)
            i = 0
        }
    }
    if i > 0 {
        for i <= 124 {
            lineimage[i] = ' '
            i++
        }
        lineprinter <- string(lineimage)
    }

    close(lineprinter)
    return
}
```

S35 Reformat

从长度为 80 的卡片上进行读取，并打印到长度为 125 个字符的 lineprinter 上。每个卡片必须跟随一个额外的空格，最后一行须使用空格进行填充。

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

```
func S35_Reformat(cardfile chan []rune, lineprinter chan string) {
    west, east := make(chan rune), make(chan rune)
    go S33_DISASSEMBLE(cardfile, west)
    go S31_COPY(west, east)
    S34_ASSEMBLE(east, lineprinter)
}
```

S36 Conway's Problem

调整前面的程序，使用「↑」替换每个成对出现的「*」。

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

```
func S35_ConwayProblem(cardfile chan []rune, lineprinter chan string) {
    west, east := make(chan rune), make(chan rune)
    go S33_DISASSEMBLE(cardfile, west)
    go S32_SQUASH_EX(west, east)
    S34_ASSEMBLE(east, lineprinter)
}
```

子程、数据表示与递归

subroutine/data representation/recursive

子程、数据表示与递归

子程是一个与用户过程并发执行的子过程：

```
[subr:SUBROUTINE || X::USER]

SUBROUTINE::[X?(value params) → …; X!(result params)]

USER::[ …; subr!(arguments); …; subr?(results)]
```

数据表示可以视为一个具有多入口的子过程，根据 guarded command 进行分支选择：

```
*[X? method1(value params) → …
□ X? method2(value params) → … ]
```

递归可以通过一个子程数组进行模拟，用户过程（零号子程）向一号子程发送必要的参数，再从起接受递归后的结果：

```
[recsub(0)::USER || recsub(i:1..reclimit):: RECSUB]

USER::recsub(1)!(arguments); … ; recsub(1)?(results);
```

S41 带余除法

编写一个类型子程，接受一个正除数与被除数，返回其商和余数。

```
[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]
```

```
func S41_DivisionWithRemainder(in chan S41_In, out chan S41_Out) {
    v := <-in
    x, y := v.X, v.Y

    quot, rem := 0, x
    for rem >= y {
        rem -= y
        quot++
    }
    out <- S41_Out{quot, rem}
}
```

S42 阶乘

给定一个上限，计算其阶乘。

```
[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 ]
```

```
func S42_Factorial(fac []chan int, limit int) {
    for i := 1; i <= limit; i++ {
        go func(i int) {
            n := <-fac[i-1]
            if n == 0 {
                fac[i-1] <- 1
            } else if n > 0 {
                // Note that here we check if i equals limit.
                // The original solution in the paper fails to terminate
                // if user input is equal or higher than the given limit.
                if i == limit {
                    fac[i-1] <- n
                } else {
                    fac[i] <- n - 1
                    r := <-fac[i]
                    fac[i-1] <- n * r
                }
            }
        }(i)
    }
}
```

S43 S44 S45 S46 集合

实现一个集合的 insert 与 has 方法

```
S::
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
]]

SEARCH::
i:integer; i := 0;
*[i<size; content(i) != n -> i:=i+1]
```

Go 实现：https://github.com/changkun/gobase/blob/f787593b4467793f8ee0b07583ea9ffde5adf2be/csp/csp.go#L392

监控与调度

Monitors / Scheduling

监控与调度

监控可以被视为与多个用户过程通信的单一过程，且总是能跟用户过程进行通信。

例如：

```
*[(i:1..100)X(i)?(value param) → …; X(i)!(results)]
```

当两个用户过程同时选择一个 X(i) 时，guarded cmd 保护了监控结果不会被错误的发送到错误的用户过程中。

S51 Buffered Channel

构造一个带缓冲的过程 X，用于平滑输出的速度（即 buffered channel）。

```
X::
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 ]
```

Go 实现：https://github.com/changkun/gobase/blob/f787593b4467793f8ee0b07583ea9ffde5adf2be/csp/csp.go#L609

S52 信号量

实现证书信号量 S，在 100 个子过程间进行共享，每个过程可以通过 V 操作在信号量非正的情况下增加信号量。

```
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]
```

Go 实现：https://github.com/changkun/gobase/blob/f787593b4467793f8ee0b07583ea9ffde5adf2be/csp/csp.go#L649

S53 哲学家进餐问题

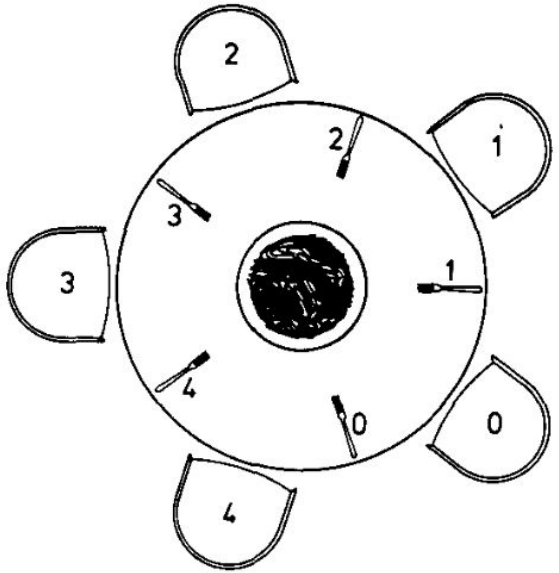
实现哲学家进餐问题。

```
PHIL = *[...during i-th lifetime ... ->
          THINK;
          room!enter();
          fork(i)!pickup();fork((i+1)mod5)!pickup();
          EAT;
          fork(i)!putdown();fork((i+1)mod5)!putdown();
          room!next()]

FORK = *[phil(i)?pickup()->phil(i)?putdown()
       □ phil((i-1)mod5)?pickup()->phil((i-1)mod5)?putdown()]

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

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



Go 实现：https://github.com/changkun/gobase/blob/f787593b4467793f8ee0b07583ea9ffde5adf2be/csp/csp.go#L746

算法

Algorithms

S61 Eratosthenes 素数筛法

实现 **Eratosthenes 素数筛法。**

```
[SIEVE(i:1..100)::
 p,mp:integer;
 SIEVE(i-1)?p;
 print!p;
 mp:=p; comment mp is a multiple of p;
*[m:integer; SIEVE(i-1)?m->
   *[m>mp->mp:=mp+p];
    [m=mp->skip □ m<mp->SIEVE(i+1)!m ]
 ]
|| SIEVE(0)::print!2; n:integer; n:=3;
      *[n<10000->SIEVE(1)!n;n:=n+2]
|| SIEVE(101)::*[n:integer;SIEVE(100)?n->print!n]
|| print::*[(i:0..101)n:integer;SIEVE(i)?n->...]
]
```

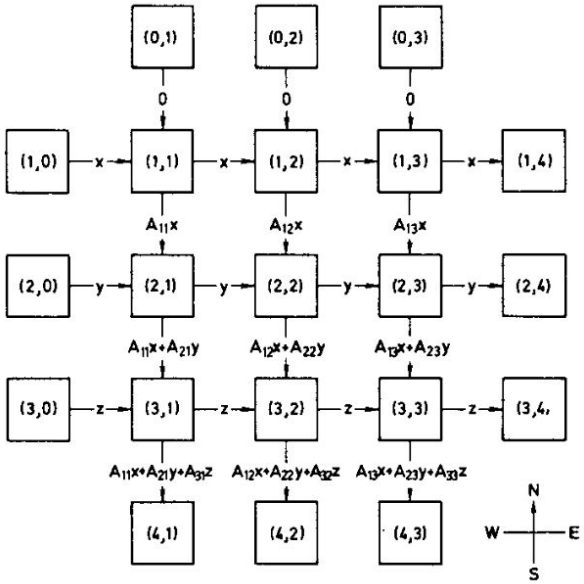
Go 实现：https://github.com/changkun/gobase/blob/f787593b4467793f8ee0b07583ea9ffde5adf2be/csp/csp.go#L833

S62 矩阵乘法

实现 3x3 矩阵乘法。

```
[M(i:1..3,0)::WEST
||M(0,j:1..3)::NORTH
||M(i:1..3,4)::EAST
||M(4,j:1..3)::SOUTH
||M(i:1..3,j:1..3)::CENTER]

NORTH = *[true -> M(1,j)!0]
EAST = *[x:real; M(i,3)?x->skip]
CENTER = *[x:real;M(i,j-1)?x->
          M(i,j+1)!x;sum:real;
          M(i-1,j)?sum;M(i+1,j)!(A(i,j)*x+sum)]
```



Go 实现：https://github.com/changkun/gobase/blob/f787593b4467793f8ee0b07583ea9ffde5adf2be/csp/csp.go#L923

反思 & 总结

Reflection

潜在的缺陷

这篇论文中讨论的 CSP 设计是 Tony Hoare 的早期提出的设计，与随后将理论完整化后的 CSP（1985）存在两大差异：

缺陷1：未对 channel 命名

并行过程的构造具有唯一的名词，并以一对冒号作为前缀：`[a::P || b::Q || … || c::R]`

在过程 P 中，命令 `b!v` 将 v 输出到名为 b 的过程。该值由在过程 Q 中出现的命令 `a?x` 输入。
过程名称对于引入它们的并行命令是局部的，并且组件过程间的通信是隐藏的。
虽然其优点是不需要在语言中引入任何 channel 或者 channel 声明的概念。

缺点：
1. 子过程需要知道其使用过程的名称，使得难以构建封装性较好的子过程库
2. 并行过程组合本质上是具有可变数量参数的运算，不能进行简化（见 CSP 1985）

缺陷2：重复指令的终止性模糊

重复指令默认当每个 guard 均已终止则指令中终止，这一假设过强。具体而言，对于 `*[a?x → P □ b?x → Q □ ...]`
要求当且仅当输入的所有过程 a,b,… 均终止时整个过程才自动终止。

缺点：

1. 定义和实现起来很复杂
2. 证明程序正确性的方法似乎比没有简单。

一种可能的弱化条件为：直接假设子过程一定会终止。

一些要点

CSP 1978 中描述的编程语言（与 Go 所构建的基于通道的 channel/select 同步机制进行对比）：

1. channel 没有被显式命名
2. channel 没有缓冲，对应 Go 中 unbuffered channel
3. buffered channel 不是一种基本的编程源语，并展示了一个使用 unbuffered channel 实现其作用的例子
4. guarded command 等价于 if 与 select 语句的组合，分支的随机触发性是为了提供公平性保障
5. guarded command 没有对确定性分支选择与非确定性（即多个分支有效时随机选择）分支选择进行区分
6. repetitive command 的本质是一个无条件的 for 循环，但终止性所要求的条件太苛刻，不利于理论的进一步形式化
7. CSP 1978 中描述的编程语言对程序终止性的讨论几乎为零
8. 此时与 Actor 模型进行比较，CSP 与 Actor 均在实体间直接通信，区别在于 Actor 支持异步消息通信，而 CSP 1978 是同步通信

…

讨论 Q&A

Q: Tony Hoare 提出 CSP 的时代背景是什么？
A: 并行计算的兴起、需要一种形式化的并行 编程语言

Q: CSP 1978 理论到底有哪些 值得我们研究的地方？

A: 一种面向并行 过程的演算代数

Q: CSP 1978 理论是否真的就是我 们目前熟知的基于通道的同步方式？

A: 不是，早期的 设计中没有 显式的对通道进行命名。

Q: CSP 1978 理论的早期设计存在什么 样的缺陷？

A: 早期设计中没有 对通道进行命名是 设计的一个主要缺陷，此外， 对程序终止性的条件描述 过于模糊，形式化程度不 够完善。

进一步阅读的参考文献

[Hoare 78] Hoare, C. A. R. (1978). Communicating sequential processes. Communications of the ACM, 21(8), 666–677.

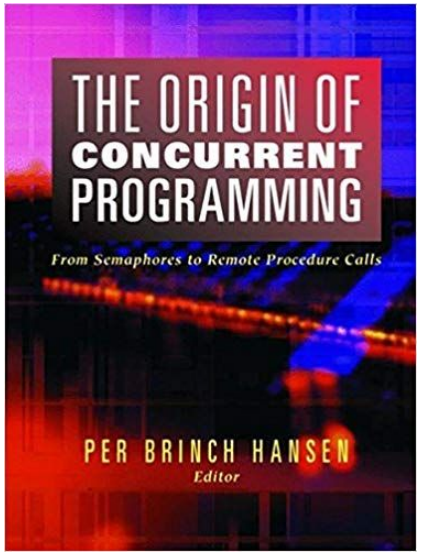
[Brookes 84] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. 1984. A Theory of Communicating Sequential Processes. J. ACM 31, 3 (June 1984), 560-599.

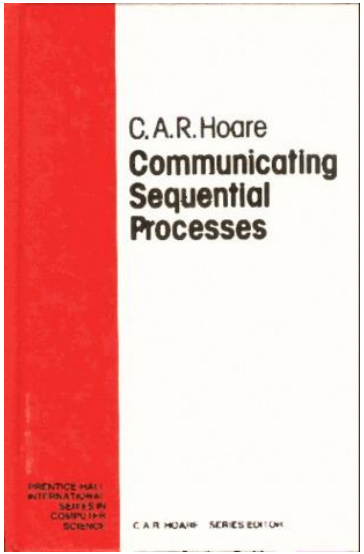
[Hoare 85] C. A. R. Hoare. 1985. Communicating Sequential Processes. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

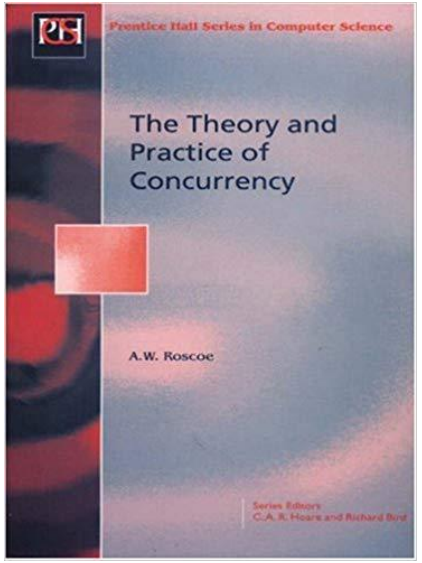
[Milner 82] R. Milner. 1982. A Calculus of Communicating Systems. Springer-Verlag, Berlin, Heidelberg.

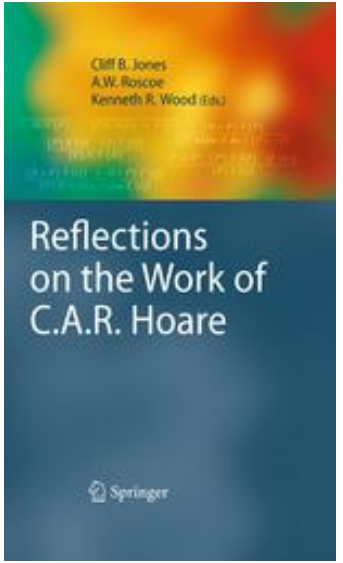
[Fidge 94] Fidge, C., 1994. A comparative introduction to CSP, CCS and LOTOS. Software Verification Research Centre, University of Queensland, Tech. Rep, pp.93-24.

推荐书籍


THE ORIGIN OF
CONCURRENT
PROGRAMMING
From Semaphores to Remote Procedure Calls
PER BRINCH HANSEN
Editor


C.A.R. Hoare
Communicating
Sequential
Processes


Prentice Hall Series in Computer Science
The Theory and
Practice of
Concurrency
A.W. Roscoe


Cliff B. Jones
A.W. Roscoe
Kenneth R. Wood (Eds.)
Reflections
on the Work of
C.A.R. Hoare
Springer