L25: JCSP

César Sánchez

Grado en Ingeniería Informática
Grado en Matemáticas e Informática
Universidad Politécnica de Madrid

Wed, 20-May-2015

Este texto se distribuye bajo los términos de la Creative Commons License
Communicating Sequential Processes (CSP)

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 exercises 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/1), coroutines (UNIX [17]), classes (SIMULA 67 [5]), processes and monitors (Concurrent Pascal [2]), clusters (CLU [13]), forms (ALPHAD [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/1, 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/1), 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.

Communications of the ACM
August 1978
Volume 21
Number 8
Message Passing (Summary)

Main idea: sequential processes that communicate sending messages.

   No shared memory!

Issues:
   (1) where to send messages?
   (2) Semantics of send and receive (blocking, arity)
Message Passing (Summary)

**Main idea:** sequential processes that communicate sending messages.

No shared memory!

**Issues:**

(1) where to send messages?
(2) Semantics of send and receive (blocking, arity)
Message Passing (Summary)

Main idea: sequential processes that communicate sending messages.

No shared memory!

Issues:
(1) where to send messages?
(2) Semantics of send and receive (blocking, arity)

Addressing

Synchronization
Message Passing (Summary)

Main idea: sequential processes that communicate sending messages.

No shared memory!

Issues:
(1) where to send messages?
(2) Semantics of send and receive (blocking, arity)

We will use by channel

We will use synchronous send

We will use 1:1 and n:1
Message Passing (Summary)

**Main idea**: sequential processes that communicate sending messages.

No shared memory!

**Issues**:
1. where to send messages?
2. Semantics of send and receive (blocking, arity)

We will use by channel

Addressing

We will use synchronous send

Synchronization

We will use 1:1 and n:1

How to implement this in Java?
JCSP

- JCSP: A Java Library for CSP

Send: ch.write(o);
Receive: o = (MyClass) ch.read(o);
JCSP: A Java Library for CSP

Send: \texttt{ch.write(o);} \\
Receive: \texttt{o = (MyClass) ch.read(o);}

casts to Object
JCSP

- JCSP: A Java Library for CSP

Send: \[\text{ch.write}(o)\];
Receive: \[o = \text{(MyClass)} \text{ch.read}(o)\];

(Sequential) Processes implement CSProcess:

```java
public class Almacen1DatoJCSP implements CSProcess {
  //...
}
```
JCSP

- JCSP: A Java Library for CSP

  Send:    ch.write(o);
  Receive: o = (MyClass) ch.read(o);

- (Sequential) Processes implement CSProcess:

  ```java
  public class Almacen1DatoJCSP implements CSProcess {
  //...
  }
  ```

- Channels:

  ![Diagram of Channels]

  - Any2OneChannel
  - One2OneChannel
JCSP

- JCSP: A Java Library for CSP

Send: \( \text{ch.write}(o); \)
Receive: \( o = \text{(MyClass)} \text{ch.read}(o); \)

- (Sequential) Processes implement CSProcess:

```java
public class Almacen1DatoJCSP implements CSProcess {
    //...
}
```

- Channels:

  - Any2OneChannel
  - One2OneChannel

- How to send and receive:

  Send: \( \text{ch.out().write}(o); \)
Receive: \( o = \text{(MyClass)} \text{ch.in().read}(o); \)
**JCSP**

- JCSP: A Java Library for CSP

  Send: `ch.write(o);`
  Receive: `o = (MyClass) ch.read(o);`

- (Sequential) Processes implement CSProcess:

  ```java
  public class Almacen1DatoJCSP implements CSProcess {
    //...
  }
  ```

- Channels:
  - Any2OneChannel
  - One2OneChannel

- How to send and receive:
  Send: `ch.out().write(o);`
  Receive: `o = (MyClass) ch.in().read(o);`
Almacen1

Almacen1

prod₁

prod₂

prod₃

cons₁

cons₂

cons₃

put(x)

hayDato?

d

get
Almacen1

Almacen1

prod1

prod2

prod3

3

put(x)

hayDato?

d

get

Almacen1

cons1

cons2

cons3
Almacen1

prod1
prod2
prod3

Almacen1

put(x)

hayDato?

get

d  3

cons1
cons2
cons3
Almacen1

Almacen1

prod₁

prod₂

prod₃

Almacen1

put(x)

get

hayDato?

d  3

cons₁

cons₂

cons₃

ch

ch
Almacen1

Almacen1

prod₁

prod₂

prod₃

put(x)

Almacen1

hayDato?

d 3

get

cons₁

cons₂

cons₃

ch

ch
while (true) {
    item = (Producto) petAlmacenar.read();
    resp = (ChannelOutput) petExtraer.read();
    resp.write(item);
}
AlmacenN

prod_1

prod_2

prod_3

AlmacenN

\(\text{put}(x)\)

get

cons_1

cons_2

cons_3
AlmacenN

prod₁

prod₂

prod₃

AlmacenN

put(x)

get

3

cons₁

cons₂

cons₃
AlmacenN

prod₁
prod₂
prod₃

AlmacenN

cons₁
cons₂
cons₃

got
put(x)
AlmacenN

AlmacenN

\( \text{prod}_1 \)

\( \text{prod}_2 \)

\( \text{prod}_3 \)

\( \text{cons}_1 \)

\( \text{cons}_2 \)

\( \text{cons}_3 \)

\( \text{put}(x) \)

\( \text{get} \)

3

ch

ch
Problem: There is no strict alternation!

Sometimes prods and cons are *both* enabled
Problem: There is no strict alternation!

Sometimes prods and cons are *both* enabled

Solution: block listening to *several* channels.
select
    x = ch1.receive();
[]
    x = ch2.receive();
[]
    x = ch3.receive();
end select

select
    when b1 => x = ch1.receive();
[]
    when b2 => x = ch2.receive();
[]
    when b3 => x = ch3.receive();
end select
Recepcion Alternativa Condicional

select
    x = ch1.receive();
[]
    x = ch2.receive();
[]
    x = ch3.receive();
end select

In CSP

select
    when b1 => x = ch1.receive();
[]
    when b2 => x = ch2.receive();
[]
    when b3 => x = ch3.receive();
end select

guards
Recepcion Alternativa Condicional in JCSP

- Recepcion alternativa no condicional:

```
services = new Alternative([...]);
services.fairselect();
```

array of AltingChannelInput
Recepción Alternativa Condicional en JCSP

- Recepción alternativa no condicional:

```java
services = new Alternative([...]);
services.fairselect();
```

array of AltingChannelInput

- Recepción alternativa condicional:

```java
services = new Alternative([...]);
services.fairselect([...]);
```

array of Booleans