**Part V. Programming Examples**

Examples of OOCF Concepts and Methodology through Programming Examples in Actalk

---

**Concept(s) Example**

**Continuation** Factorial

**Divide & Conquer and Join Continuation**
- MultiplyInRange

**Pipeline** Prime Numbers

**Synchronization** Bounded Buffer

---

**Computing Factorial with Continuations**

Recursive computation of $\text{factorial}(n)$ is split between:
- the recursive call $\text{factorial}(n-1)$,
- multiplying that value by $n$ and returning it (to the current reply destination)

This latter task is delegated to a continuation which encapsulates:
- the program of the remaining computation (multiply $\text{factorial}(n-1)$ by $n$, and return the result to current reply destination),
- the context of the current computation needed to resume it later ($n$ and the current reply destination)

---

**Factorial in Scheme (lexical scope Lisp)**

```scheme
; factorial with recursion
(define (factorial-r n)
  (if (= n 0)
      1
      (* (factorial-r (- n 1)) n)))

? (factorial-r 10)
= 3628800

; factorial with continuation
(define (factorial-c n c)
  (if (= n 0)
      (c 1)
      (factorial-c (- n 1) 
                   (lambda (v) (c (* v n))))))

? (factorial-c 10 (lambda (v) v))
= 3628800
```

---

**Factorial**

![Factorial Diagram]

---
class Factorial

ActiveObject: #Factorial
instanceVariableNames: 
classVariableNames: 
poolDictionaries: 
category: 'Tutorial-Examples'!

!Factorial methodsFor: 'script'!
n: n replyTo: r
  n = 0
  ifTrue: [r reply: 1]
  ifFalse: [aself n: n-1 replyTo: (FactorialContinuation new n: n r: r) active]!

!Factorial class methodsFor: 'example'!
example
  "Factorial example"
  | aFac |
  aFac := Factorial new active.
  aFac n: 10 replyTo: Print.
  aFac n: 5 replyTo: Print!

ActiveObject subclass: #FactorialContinuation
instanceVariableNames: 'n r '
classVariableNames: 
poolDictionaries: 
category: 'Tutorial-Examples'!

!FactorialContinuation methodsFor: 'initialization'!
n: anInteger r: aContinuation
  n := anInteger.
  r := aContinuation!

!FactorialContinuation methodsFor: 'script'!
reply: v
  r reply: n * v! 

Advantages of Continuation

asynchronous + continuation
vs synchronous call

in case of divergence
  ex: factorial(-1)
  availability remains,
  but eventually no more resources (memory full!)

recursion is possible
  with synchronous communication, recursion implies deadlock!

However systematic programming with continuations is too low level. They may be automatically generated from a higher level language by a compiler (actor approach, see Part VI).

Levels of Concurrency

In fact the computation of factorial actually remains sequential!

Concurrency occurs within simultaneous requests

We will now describe another algorithm managing concurrent sub-computations

The idea is to decompose computation into sub-computations executed concurrently with recombination of partial sub-results
Factorial with Recursive Partition

Computation of N! is equivalent to multiplying all numbers in interval [1, N].

These multiplications may occur concurrently. The idea is to divide the interval [1, N] into two sub-intervals:

\[[1 \text{ mid}] \text{ and } [\text{mid}+1, N]\]

(where mid = \(1+N/2\))

and to perform the two sub-computations concurrently.

Computation is recursive and creates two new active objects to compute these two sub-intervals with a join continuation as the common reply destination.

This join continuation will multiply the two partial results.

Factorial with Divide and Conquer

[1, N] \[1, p\] \[p+1, N\]

\[[1, 3]\] \[3, 3\] \[N-1, N\]

\[[1, 1]\] \[2, 2\] \[N-1, N\]

Join continuations recombine partial results.

A join continuation is equivalent to a data-flow operator, which waits for two (or more) values to recombine, and sends the result to the current continuation.

MultiplyInRange

ActiveObject subclass: #MultiplyInRange
instanceVariableNames: ''
classVariableNames: ''
poolDictionaries: ''
category: 'Tutorial-Examples''

!MultiplyInRange methodsFor: 'script'!
from: i to: n replyTo: r
| jc mid | "[i, n]"
| = n ifTrue: [r reply: i]
ifFalse:
  [jc := (MultiplyJoinContinuation new r: r) active]
  mid := (i+n)/2.
  MultiplyInRange new "[i, mid]"
  from: i to: mid replyTo: jc.
  MultiplyInRange new "[mid+1, n]"
  from: mid+1 to: n replyTo: jc! !

MultiplyInRange class
instanceVariableNames: ''

!MultiplyInRange class methodsFor: 'example'!
example "MultiplyInRange example"
MultiplyInRange new active
from: 1 to: 10 replyTo: Print! !
Concurrent Computation/Problem Solving

Three main classes of concurrent algorithms for problem solving are:

# partition

the problem is decomposed in sub-problems, partial results are recombined afterwards

# pipeline

the problem is decomposed by linking up computations through which data flow up

# cooperation

the problem is decomposed in a collection of entities (agents) which will themselves organize decomposition, allocation, and coordination of their tasks (see multi-agent systems, Part IX)

---

class PrimeFilter

ActiveObject subclass: #PrimeFilter
instanceVariableNames: 'n next'
classVariableNames: ''
poolDictionaries: ''
category: 'Tutorial-Examples'

PrimeFilter subclass: PrimeFilter

instanceVariableNames: n next
classVariableNames: ''
poolDictionaries: ''
category: 'Tutorial-Examples'

PrimeFilter methodsFor: 'script'

n := aPrimeNumber
next := (PrimeFilter new n: i) active

PrimeFilter methodsFor: 'initializing'

category: 'Tutorial-Examples'
poolDictionaries: ''
classVariableNames: ''
instanceVariableNames: 'n next'
ActiveObject subclass: #PrimeFilter

class MultiplyJoinContinuation

ActiveObject subclass: #MultiplyJoinContinuation
instanceVariableNames: 'v1 c'
classVariableNames: ''
poolDictionaries: ''
category: 'Tutorial-Examples'

MultiplyJoinContinuation methodsFor: 'initialization'

c: aContinuation

c := aContinuation! !

MultiplyJoinContinuation methodsFor: 'script'

reply: v

ifTrue: [v1 := v] "memorize first value"
inFalse: [c reply: v1*v] ! ! "otherwise, compute"

v1 isNil                "if no value received yet"

The behavior of the join continuation changes serially:
1) memorize value
2) combine (multiply) it with the first value and return the result to the continuation

A test is necessary to check if a first value has already been accepted. This problem will be solved gracefully in the actor model of computation (see Part VII).

---

Pipeline: Computation of Prime Numbers

We compute prime numbers through an ordered chain of sieves

Each sieve represents a prime number already found

Successive integers are sent through the chain, each filter testing if it divides the integer

If an integer reaches successfully the last filter of the chain, a new prime number has been found, it will then be added at the end of the chain

---

Programming Methodology and Examples in Actalk

J.-P. Briot  Examples in Actalk  Page 13

J.-P. Briot  Examples in Actalk  Page 16
### Synchronization: Producer/Consumer with a Bounded Buffer

A producer and consumer exchange data through a bounded buffer.

The issue is to synchronize production and consumption to the availability (fullness or emptiness) of the buffer (e.g., disable put requests while the buffer is full).

---

### class BoundedBufferActivity

(Abstract States model of synchronization)

```actalk
class BoundedBufferActivity
  AbstractStatesActivity subclass: #ASBBActivity
  instanceVariableNames: ""
  classVariableNames: ""
  poolDictionaries: ""
  category: 'Tutorial-Examples'!

  !ASBBActivity methodsFor: 'abstract states'
  empty
    ^#(put:):
  full
    ^(self empty) + (self full)!

  initialAbstractState
    ^#empty!

  partial
    [oself is empty ifTrue: [ #empty ]
     ifFalse: [ oself isFull
                  ifTrue: [ #full ]
                  ifFalse: [ #partial ] ] ]
```

---

### class Producer

```actalk
class Producer
  ActiveObject subclass: #Producer
  instanceVariableNames: 'buffer delay'
  classVariableNames: ""
  poolDictionaries: ""
  category: 'Tutorial-Examples'!

  !Producer methodsFor: 'initializing'
  buffer: aBoundedBuffer delay: seconds
    buffer := aBoundedBuffer.
    delay: seconds! !

  !Producer methodsFor: 'script'
  run: max
    1 to: max do: [ :i | 
      buffer put: i. 
      (Delay forSeconds: delay) wait] !
```

---

### class Consumer

```actalk
class Consumer
  ActiveObject subclass: #Consumer
  instanceVariableNames: 'buffer delay'
  classVariableNames: ""
  poolDictionaries: ""
  category: 'Tutorial-Examples'!

  !Consumer methodsFor: 'initializing'
  buffer: aBoundedBuffer delay: seconds
    buffer := aBoundedBuffer.
    delay: seconds! !

  !Consumer methodsFor: 'script'
  run: max
    max timesRepeat:
      [buffer get.
       (Delay forSeconds: delay) wait] !
```
Monitors (on Passive Objects)

**Activation Synchronization**

\[ \text{activation synchronization} = \text{mutual exclusion} + \text{service synchronization} \]

(suspension onto conditions)

Service synchronization is expressed with conditions (variables: Hoare, or expressions: Kessel) on which processes will be suspended while conditions are not fulfilled.

![Diagram of a monitor with conditions and synchronization types](image-url)