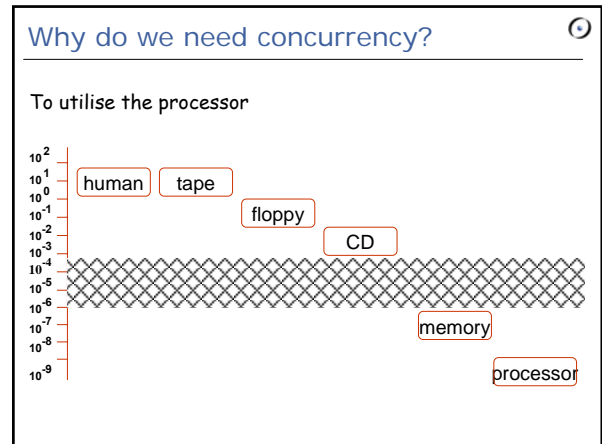


Software Architecture  
 Bertrand Meyer  
 ETH Zurich, March-July 2007

**Lecture 9: Concurrency & SCOOP**

(with material by Piotr Nienaltowski & Volkan Arslan)

Chair of Software Engineering



### Concurrent Programming

Definition (Ben- Ai, 1982):

Programming notations and techniques for **expressing potential parallelism** and solving the resulting **synchronization and communication** problems

Provides an **abstract setting** to study parallelism without getting into implementation details

### Why do we need concurrency?

- Multiprogramming on a single computer
- Distributed programming across networks
- Multiple activities in one tool (e.g. mail client, Web browser, IDE)
- At the hardware level: multicore architectures

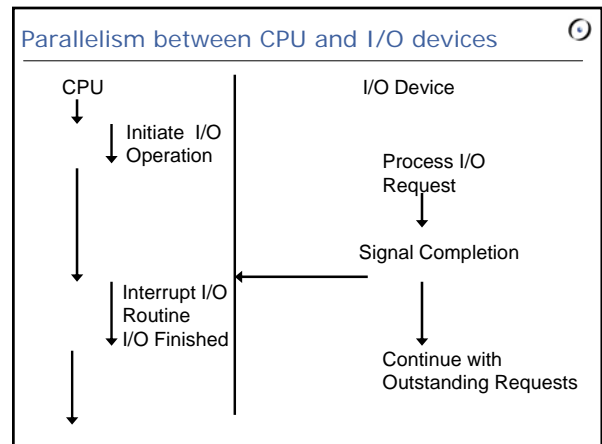
### Terminology

Dijkstra (1968)  
 A concurrent program is a collection of autonomous sequential processes, executing (logically) in parallel

Each process has a single thread of control

Implementation can multiplex process execution in three ways:

- > **Multiprogramming:** single processor
- > **Multiprocessing:** several processors with shared memory
- > **Distributed Processing:** several processors not sharing memory



## Processes and threads

All operating systems provide processes  
 Each process executes in its own virtual machine (VM) to avoid interference from others  
 Also in modern OSes: several **threads** within one VM. Like lighter versions of processes, but:

- Unrestricted access to respective VM
- Language & programmer must avoid interference

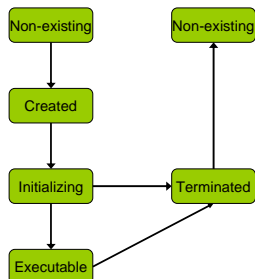
Language may define concurrency or leave it to the OS:

- Ada, Java and C#, Eiffel with SCOOP, provide concurrency
- C, C++ do not

## Processes and Objects

**Active** objects — undertake spontaneous actions  
**Reactive** objects — only perform actions when invoked  
**Resources** — reactive but can control order of actions  
**Passive** — reactive, but no control over order  
**Protected** resource — passive resource controller  
**Server** — active resource controller

## Process states



## Process Representation

Coroutines  
 Fork and Join  
 Cobegin  
 Explicit Process Declaration

## Concurrent Programming Constructs

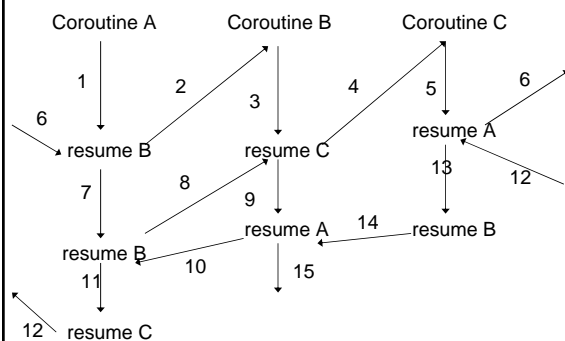
Concurrent programming language mechanisms support:

- Expressing concurrent execution through the notion of process (or/and threads)
- Process synchronization
- Inter-process communication

Processes may be

- Independent
- Cooperating
- Competing

## Coroutine Flow Control



## Fork and join

(See UNIX/POSIX and programming languages such as Mesa.)

```
function F return is ...;
procedure P;
...
C := fork F;
...
J := join C;
...
end P;
```

**Fork:** designated routine starts executing concurrently with invoker  
**Join:** invoker waits for completion of invoked routine

After fork, P and F will be executing concurrently. At join, P will wait until F has finished (if not already done)

## Tasks and Ada

The unit of concurrency in Ada is called a task

Tasks properties:

- Explicitly declared: no fork/join, COBEGIN etc.
- Declared at any program level; created implicitly on entry to declaration scope or via allocator
- Communicate and synchronise via various mechanisms: rendezvous (synchronised message passing), protected units (monitor/conditional critical region), shared variables

## Cobegin

cobegin (or parbegin or par) is a structured way of denoting concurrent execution of instructions:

```
cobegin
  S1;
  S2;
  S3;
  .
  .
  Sn
coend
```

Terminates when all have terminated

Example languages: Edison, occam2.

## Robot Arm example

```
type Dimension is (Xplane, Yplane, Zplane);
task type Control(Dim : Dimension);
C1 : Control(Xplane);
C2 : Control(Yplane);
C3 : Control(Zplane);

task body Control is
  Position : Integer;    -- absolute position
  Setting  : Integer;    -- relative movement
begin
  Position := 0;        -- rest position
  loop
    New_Setting (Dim, Setting);
    Position := Position + Setting;
    Move_Arm (Dim, Position);
  end loop;
end Control;
```

## Explicit process Declaration

The structure of a program can be made clearer if routines state whether they will be executed concurrently

Note that this does not say when they will execute

```
task body Process is
begin
  . . .
end;
```

Languages that support explicit process declaration may have explicit or implicit process/task creation

## Java threads

- Dynamic thread creation
- Constructors allow arbitrary data as arguments
- No master or guardian concept; garbage collection cleans up no longer accessible objects
- Main program terminates when all user threads have terminated
- One thread can wait for another to terminate through join
- isAlive allows a thread to determine if another has terminated

## Synchronization and Communication

The **correct behaviour of a concurrent program** depends on **synchronisation** and **communication** between its processes

**Synchronisation**: the satisfaction of constraints on interleaving of process actions (e.g. action by a process should only occur after action by another)

Also: bring two processes simultaneously into predefined states

**Communication**: the passing of information from one process to another

## Shared resource communication

```
type Coordinates is
  record
    X : Integer;
    Y : Integer;
  end record;
Shared_Coordinate: Coordinates;
```

```
task body Helicopter is
  Next: Coordinates;
begin
  loop
    Compute_New_Coordinates(Next);
    Shared_Coordinate := Next;
  end loop
end;
```

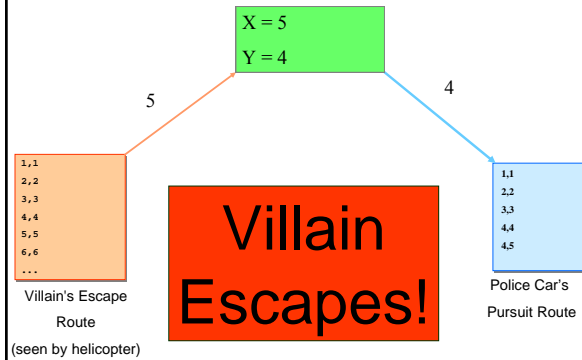
```
task body Police_Car is
begin
  loop
    Plot(Shared_Coordinates);
  end loop;
end;
```

## Synchronization and Communication

Concepts are linked since communication **requires** synchronisation, and synchronisation can be **considered as content-less** communication.

Data communication is usually based upon either **shared variables** or **message passing**.

## Villain's escape route



## Shared Variable Communication

Examples: busy waiting, semaphores and monitors  
Unrestricted use of shared variables is unreliable and unsafe due to multiple update problems

Consider two processes updating a shared variable,  $x$ , with the assignment:  $x := x + 1$

- load the value of  $x$  into some register
- increment the value in the register by 1 and
- store the value in the register back to  $x$

As the three operations are **not indivisible**, two processes simultaneously updating the variable could follow an interleaving that would **produce an incorrect result**

## Avoiding interference

The parts of a process that access shared variables must be executed **indivisibly (atomically)** with respect to each other

These parts are called **critical sections**

The required protection is called **mutual exclusion**

## Mutual exclusion

A sequence of instructions that must appear to be **executed indivisibly** is called a **critical section**

The synchronisation required to protect a critical section is known as **mutual exclusion**

**Atomicity** is assumed to be present at the memory level. If one process is executing  $x := 5$ , simultaneously with another executing  $x := 6$ , the result will be either 5 or 6 (not some other value)

If two processes are updating a structured object, this atomicity will only apply at the **single word element level**

## Busy waiting (spinning)

### Busy waiting and condition synchronizing

```
process P1; (* waiting process *)
  while flag = down do
    null
  end
end P1;
```

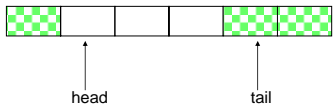
```
process P2; (* signalling process *)
  flag = up;
end P2;
```

## Condition synchronisation

Needed when a process wishes to perform an operation that can only be performed safely if another process has itself taken some action or is in some defined state

E.g. a bounded buffer has 2 condition synchronisation:

- the producer processes must not attempt to deposit data onto the buffer if the buffer is full
- the consumer processes cannot be allowed to extract objects from the buffer if the buffer is empty



## Busy waiting

### Busy waiting and mutual exclusion

```
process P;
  loop
    entry protocol
    critical section
    exit protocol
    non critical section
  end
end P;
```

## Busy waiting

One way to implement synchronisation is to have processes set and check shared variables that are acting as flags  
Works well for condition synchronisation but no simple method for mutual exclusion

Inefficient:

- Processes uses up computing cycles when they cannot perform useful work
- On a multiprocessor system, can give rise to excessive traffic on the memory bus or network

## Busy waiting and mutual exclusion (not correct)

```
process P1;
  loop
    flag1 := up (* announce intent to enter *)
    while flag2 = up do
      null (* busy wait if the other process is in *)
    end; (* its critical section *)
    <critical section>
    flag1 := down (* exit protocol *)
    <non critical section>
  end
end P1;
```

```
process P2;
  loop
    flag2 := up
    while flag1 = up do
      null
    end;
    <critical section>
    flag2 := down
    <non critical section>
  end
end P2;
```

## Interleaving of P1 and P2

P1 sets its flag (flag1 = up)  
P2 sets its flag (flag2 = up)  
P2 checks flag1 (it is up therefore P2 loops)  
**P2 enters its busy wait**  
P1 checks flag2 (it is up therefore P1 loops)  
**P1 enters its busy wait**

### Result

- Both P1 and P2 will remain in their busy waits
- Neither can get out because the other cannot get out  
→ **LiveLock**

## Semaphores

Dijkstra (1968)

Simple mechanism for programming

- Mutual exclusion
- Condition synchronisation

Benefits:

- Simplify the protocols for synchronisation
- Remove the need for busy-wait loops

## Busy waiting and mutual exclusion (not correct)

```
process P1;
loop
  while flag2 = up do
    null (* busy wait if the other process is in *)
  end;
  (* its critical section *)
  flag1 := up (* announce intent to enter *)
  <critical section>
  flag1 := down (* exit protocol *)
  <non critical section>
end
end P1;
```

```
process P2;
loop
  while flag1 = up do
    null
  end;
  flag2 := up
  <critical section>
  flag2 := down
  <non critical section>
end
end P2;
```

## Semaphores

Non-negative integer variable

Two operations apart from initialization:

- wait (S) (originally known as P (S))
  - If the value of S > 0 then decrement its value by one; otherwise delay process until S > 0 (and then decrement its value).
- signal (S) (originally known as V (S))
  - Increment the value of S by one.

Both are **atomic** (indivisible). Two processes executing wait on same semaphore **cannot interfere** and cannot fail

## Interleaving of P1 and P2

P1 and P2 are in their non-critical section  
flag 1 = flag 2 = down  
P1 checks flag2 (flag2 = down)  
P2 checks flag1 (flag1 = down)  
P2 sets its flag (flag2 = up)  
**P2 enters critical section**  
P1 sets its flag (flag1 = up)  
**P1 enters critical section**  
  
**P1 and P2 are both in their critical section!**

## Condition synchronisation

```
var consyn : semaphore (* init 0 *)
```

```
process P1;
(* waiting process *)
instruction X;
wait (consyn)
instruction Y;
end P1;
```

```
process P2;
(* signalling proc *)
instruction A;
signal (consyn)
instruction B;
end P2;
```

In what order will the instructions execute?

## Mutual exclusion (mutex)

```
(* mutual exclusion *)
var mutex : semaphore; (* initially 1 *)
```

```
process P1;
instruction X
wait (mutex);
instruction Y
signal (mutex);
instruction Z
end P1;
```

```
process P2;
instruction A;
wait (mutex);
instruction B;
signal (mutex);
instruction C;
end P2;
```

In what order will the instructions execute?

## SCOOP in a nutshell

- No intra-object-concurrency
- One keyword: **separate**, indicates thread of control is "elsewhere"
- Reserve one or more objects through argument passing
- Preconditions become wait conditions
- Exception-based mechanism to break lock

## SCOOP

SCOOP: Simple Concurrent Object-Oriented Programming

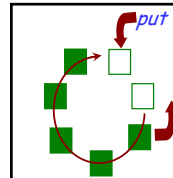
First iteration 1990 -- CACM, 1993

*Object-Oriented Software Construction*, 2<sup>nd</sup> edition, 1997

Prototype implementations, 1995-now

Now being done for good at ETH

On top of Eiffel Software's compiler and EiffelThreads library (native Windows, .NET, Unix, Linux...)

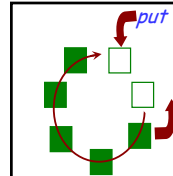


Chair of Software Engineering

ETH

## SCOOP: The basic goal

Can we bring concurrent programming to the same level of abstraction and convenience as sequential programming?



```
put (b: BUFFER[G]; v: G)
-- Store v into b.
require
not b.is_full
do
...
ensure
not b.is_empty
end
```

```
my_queue: BUFFER[T]
...
if not my_queue.is_full then
    put(my_queue, t)
end
```



Chair of Software Engineering

ETH

## The issue

Can we bring concurrent programming to the same level of abstraction and convenience as sequential programming?

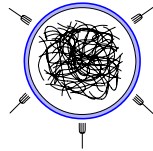
## Previous advances in programming

	"Structured programming"	"Object technology"
Use higher-level abstractions	✓	✓
Helps avoid bugs	✓	✓
Transfers tasks to implementation	✓	✓
Lets you do stuff you couldn't before	NO	✓
Removes restrictions	NO	✓
Adds restrictions	✓	✓
Has well-understood math basis	✓	✓
Doesn't require understanding that basis	✓	✓
Permits less operational reasoning	✓	✓

## Dining philosophers

```

class PHILOSOPHER inherit
  PROCESS
  rename
    setup as getup
  redefine step end
feature {BUTLER}
  step
  do
    think; eat(left, right)
  end
  eat(l, r: separate FORK)
  -- Eat, having grabbed l and r.
  do ... end
end
    
```



## Then and now

### Sequential programming:

Used to be messy  
Still hard but:

- > Structured programming
- > Data abstraction & object technology
- > Design by Contract
- > Genericity, multiple inheritance
- > Architectural techniques

Switch from operational reasoning to logical deduction (e.g. invariants)

### Concurrent programming:

Used to be messy  
**Still messy**

Example: threading models in most popular approaches

Development level: sixties/seventies

Only understandable through operational reasoning

## Data races and other delights of life

Source: Christopher von Praun, Thomas Gross, *Journal of Object Technology*, 2005

```

class ResourceStoreManager {
  boolean closed = false;
  Map entries = new HashMap();

  synchronized void checkClosed() {
    if (closed)
      throw new RuntimeException();
  }

  ResourceStore loadResourceStore(...) {
    checkClosed();
    StoreEntry se = lookupEntry(...);
    return se.getStore();
  }
}
    
```

```

synchronized Entry lookupEntry(...) {
  Entry e = (Entry) entries.get(...);
  if (e == null) {
    e = new Entry();
    entries.put(..., e);
  }
  return e;
}

synchronized void shutdown() {
  while (...) {
    // remove all entries
  }
  closed = true;
}
    
```

## Can object technology help?

"*Objects are naturally concurrent*" (Milner)

Many attempts, often based on (self-contradictory) notion of "Active objects"

Often lead to "Inheritance anomaly"

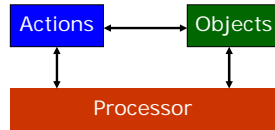
None widely accepted

**In practice:** low-level mechanisms on top of O-O language

## Object-oriented computation

To perform a computation is

- > To apply certain **actions**
- > To certain **objects**
- > Using certain **processors**



## Reasoning about objects

$$\{INV \text{ and } Pre_r\} \text{ body}_r \{INV \text{ and } Post_r\}$$


---


$$\{Pre_r\} x.r(a) \{Post_r\}$$

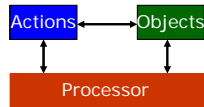
## What makes an application concurrent?

### Processor:

Thread of control supporting sequential execution of instructions on one or more objects

Can be implemented as:

- > Computer CPU
- > Process
- > Thread
- > AppDomain (.NET) ...



Will be mapped to computational resources

## Reasoning about objects

Only  $n$  proofs if  $n$  exported routines!

$$\{INV \text{ and } Pre_r\} \text{ body}_r \{INV \text{ and } Post_r\}$$


---

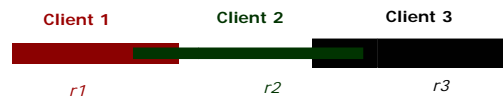

$$\{Pre_r\} x.r(a) \{Post_r\}$$

## Handling rule

All calls on an object are executed by the object's handler

## In a concurrent context

Only  $n$  proofs if  $n$  exported routines?



**No overlapping!**

$$\{INV \text{ and } Pre_r\} \text{ body}_r \{INV \text{ and } Post_r\}$$


---


$$\{Pre_r\} x.r(a) \{Post_r\}$$

## Mutual exclusion rule

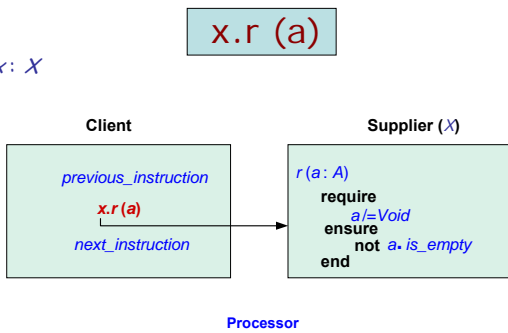
At most one feature may execute on any one object at any one time

## Separateness rule

Calls on non separate objects are blocking  
Call on separate objects are non blocking

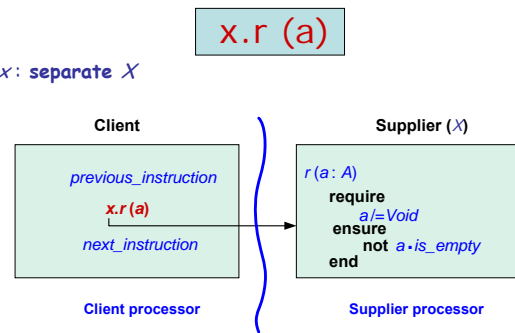
## Feature call: sequential

$x: X$



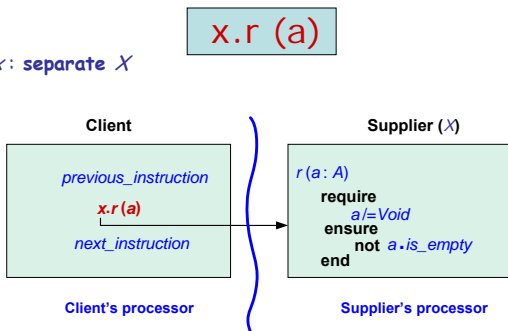
## Feature call: asynchronous

$x: \text{separate } X$



## Feature call: asynchronous

$x: \text{separate } X$



## The fundamental difference

To wait or not to wait:

If same processor, synchronous

If different processor, asynchronous

Difference must be captured by syntax:

- $x: X$
- $x: \text{separate } X$  -- potentially different processor

### Consistency: avoiding traitors

```
class C feature
  nonsep: SOME_TYPE
  sep: separate SOME_TYPE
  nonsep := sep
  nonsep.p (a)
end
```

Traitor!

### Consistency

```
Client:
class C feature
  a: SOME_TYPE
  sep: separate B
  sep.p (a)
end

Supplier:
class B feature
  p (a: separate SOME_TYPE)
  is do ... a.g ...
end
end
```

### No-traitors rule

If the source of an attachment is separate, so must the target be.

Attachment: assignment or argument passing)

### Separateness consistency rule

For any reference actual argument in a separate call, the corresponding formal argument must be declared as separate

Separate call:  $a.f(\dots)$  where  $a$  is separate

### Consistency

```
Client:
class C feature
  a: SOME_TYPE
  sep: separate B
  sep.p (a)
end

Supplier:
class B feature
  p (a: SOME_TYPE)
  is do ... a.g ...
end
end
```

### If no access control

```
my_stack: separate STACK[T]
...
my_stack.push (a)
y := my_stack.top
```

## Access control policy

Require target of separate call to be formal argument of enclosing routine:

```
put (stack: separate STACK[T]; value: T)
  -- Push value on top of stack.
do
  stack.push(value)
end
```

## Wait rule

A routine call with separate arguments will execute when all corresponding processors are available

and hold them exclusively for the duration of the routine

## Access control policy

Target of a separate call must be formal argument of enclosing routine:

```
store(buffer: separate BUFFER[T]; value: T)
  -- Store value into buffer.
do
  buffer.put(value)
end
```

To use separate object:

```
my_buffer: separate BUFFER[INTEGER]
create my_buffer
store(my_buffer, 10)
```

## Contracts in Eiffel

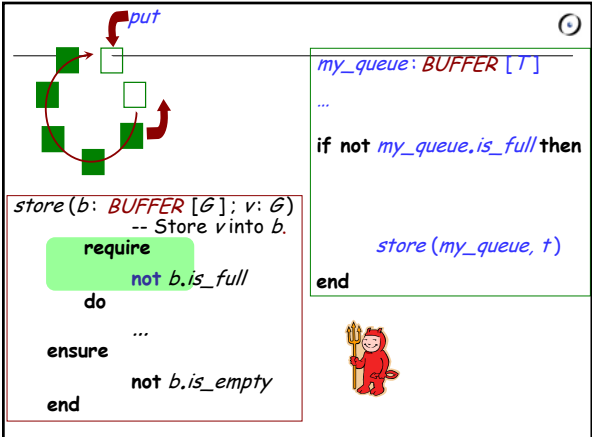
```
store(buffer: BUFFER[INTEGER]; v: INTEGER)
  -- Store v into buffer.
require
  not buffer.is_full
  v > 0
do
  buffer.put(v)
ensure
  not buffer.is_empty
end
...
store(my_buffer, 10)
```

Precondition

## Separate argument rule

The target of a separate call must be an argument of the enclosing routine


Separate call:  $x.f(\dots)$  where  $x$  is separate



The diagram shows a circular queue with five slots. A red arrow labeled 'put' points to the top slot. Below the diagram is a code snippet for a 'store' routine that uses a 'separate' argument.

```
my_queue: BUFFER[T]
...
if not my_queue.is_full then
  store(my_queue, t)
end
```

```
store(b: BUFFER[G]; v: G)
  -- Store v into b.
require
  not b.is_full
do
  ...
ensure
  not b.is_empty
end
```



## From preconditions to wait-conditions

```
store (buffer: separate BUFFER [INTEGER] ; v: INTEGER)
  -- Store v into buffer.
  require
    not buffer.is_full
    v > 0
  do
    buffer.put (v)
  ensure
    not buffer.is_empty
  end
  ...
  store (my_buffer, 10)
```

Precondition becomes **wait condition**

## Resynchronization

No explicit mechanism needed for client to resynchronize with supplier after separate call.

The client will wait only when it needs to:

```
x.f
x.g(a)
y.f
...
value := x.some_query
```

Wait here!

Lazy wait (Denis Caromel, wait by necessity)

## Separate precondition rule

A precondition causes the client to wait

## Resynchronization rule

Clients wait for resynchronization on queries

## Full synchronization rule

- A call with separate arguments waits until:
- The corresponding objects are all available
  - Preconditions hold

```
x.f(a)
where a is separate
```

## Semantics of postconditions

```
class CLIENT
  feature
    york, tokyo: separate LOCATION
    ...
    spawn_two_activities (I1, I2: separate LOCATION)
      do
        I1.do_job
        I2.do_job
      ensure
        I1.is_ready
        I2.is_ready
      end
    ...
    spawn_two_activities (york, tokyo)
    do_local_stuff
    get_result (york)
    ...
  end
```

Asynchronous evaluation; processor(s) available when and if postcondition holds.

Each clause evaluated individually.

Wait for york only.

## Generalised semantics of postconditions

> Each locked processor released only when related postcondition clauses hold.

> Each postcondition clause is evaluated individually.

**ensure**

*location\_1.is\_ready*  
*location\_2.is\_ready*

is different from

**ensure**

*location\_1.is\_ready* **and** *location\_2.is\_ready*

> This semantics boils down to correctness semantics for non-separate postconditions.

## Duels

Library features

Challenger →	<i>normal_service</i>	<i>immediate_service</i>
↓ Holder		
<i>retain</i>	Challenger waits	Exception in challenger
<i>yield</i>	<b>Challenger waits</b>	Exception in holder; serve challenger

## Duels

Library features

Challenger →	<i>normal_service</i>	<i>immediate_service</i>
↓ Holder		
<i>retain</i>	Challenger waits	Exception in challenger
<i>yield</i>	Challenger waits	Exception in holder; serve challenger

## Duels

Library features

Challenger →	<i>normal_service</i>	<i>immediate_service</i>
↓ Holder		
<i>retain</i>	Challenger waits	<b>Exception in challenger</b>
<i>yield</i>	Challenger waits	Exception in holder; serve challenger

## Duels

Library features

Challenger →	<i>normal_service</i>	<i>immediate_service</i>
↓ Holder		
<i>retain</i>	<b>Challenger waits</b>	Exception in challenger
<i>yield</i>	Challenger waits	Exception in holder; serve challenger

## Duels

Library features

Challenger →	<i>normal_service</i>	<i>immediate_service</i>
↓ Holder		
<i>retain</i>	Challenger waits	Exception in challenger
<i>yield</i>	Challenger waits	<b>Exception in holder; serve challenger</b>

## Other aspects

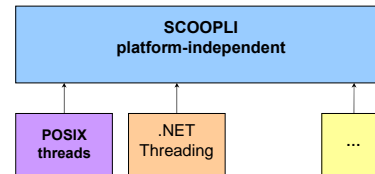
>What if a separate call, e.g. in

```
r(a separate T)
do
  a.f
  a.g
  a.h
end
```

cause an exception?

## Implementation: two-level architecture

Adaptable to many environments  
Currently implemented for native Windows  
(using POSIX threads) and .NET



## Refined proof rule (partial correctness)

$$\frac{\{INV_r \wedge Pre_r(x)\} body_r \{INV_r \wedge Post_r(x)\}}{\{Pre_r(a^{targ})\} e.r(a) \{Post_r(a^{targ})\}}$$

Hoare-style "sequential" reasoning applies to synchronous and asynchronous calls

Targettable expressions are:

- > Attached (statically known to be non-void)
- > Handled by processor locked in current context

Targettability known statically from type

## SCOPLI: Library for SCOOP

Library-based solution

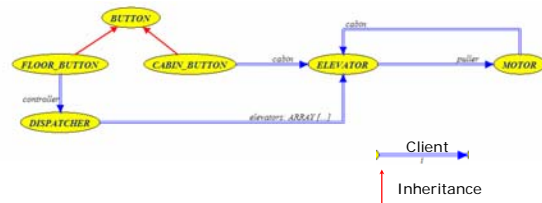
Implemented in Eiffel

Preprocessor and type checker

## Example: asynchronous calls

```
store_two(buf: separate BUFFER {INTEGER}; i, j: INTEGER)
require
  buf.count <= buf.capacity - 2
do
  {buf.count <= buf.capacity - 2}
  buf.put(i)
  {buf.count = old buf.count + 1 &
   buf.count <= buf.capacity - 1}
  buf.put(j)
  {buf.count = old buf.count + 2 &
   buf.count <= buf.capacity}
ensure
  buf.count = old buf.count + 2
end
```

## Elevator example architecture



For maximal concurrency, all objects are separate

```

Class BUTTON
class
    BUTTON
feature
    target: INTEGER
end

```

```

Class ELEVATOR
feature {NONE} -- Implementation
    process_request
        -- Handle next pending request, if any.
        local floor: INTEGER do
            if not pending.is_empty then
                floor := pending.item ; actual_process(puller, floor)
                pending.remove
            end
        end
    end

    actual_process(m: separate MOTOR; floor: INTEGER)
        -- Handle next pending request, if any.
        do
            moving := true ; m.move(floor)
        end
end

feature {NONE} -- Implementation
    puller: separate MOTOR ; pending: QUEUE[INTEGER]
end

```

```

Class CABIN_BUTTON
class CABIN_BUTTON inherit
    BUTTON
feature
    cabin: separate ELEVATOR

    request
        -- Send to associated elevator a request to stop on level
        target.
        do
            actual_request(cabin)
        end

    actual_request(e: separate ELEVATOR)
        -- Get hold of e and send a request to stop on level target.
        do
            e.accept(target)
        end
end

```

```

Class MOTOR
class MOTOR feature {ELEVATOR}
    move(floor: INTEGER)
        -- Go to floor; once there, report.
        do
            gui_main_window.move_elevator(cabin_number, floor)
            signal_stopped(cabin)
        end
    end

    signal_stopped(e: separate ELEVATOR)
        -- Report that elevator e stopped on level position.
        do e.record_stop(position) end
feature {NONE}
    cabin: separate ELEVATOR ; position: INTEGER -- Current floor
level.
    gui_main_window: GUI_MAIN_WINDOW
end

```

```

Class ELEVATOR
class ELEVATOR feature {BUTTON, DISPATCHER}
    accept(floor: INTEGER)
        -- Record and process a request to go to floor.
        do
            record(floor)
            if not moving then process_request end
        end
end

feature {MOTOR}
    record_stop(floor: INTEGER)
        -- Record information that elevator has stopped on
        floor.
        do
            moving := False ; position := floor ; process_request
        end
end

```

```

Why SCOOP?

SCOOP model
> Simple yet powerful
> Easier and safer than common concurrent techniques, e.g. Java Threads
> Full concurrency support
> Full use O-O and Design by Contract
> Supports various platforms and concurrency architectures
> One new keyword: separate

Tools
> SCOOPLI library
> Pre-processor and type checker
> Full integration with the compiler coming soon

```

## Why SCOOP?

Extend object technology with **general** and **powerful** concurrency support

Provide the industry with simple techniques for parallel, distributed, internet, real-time programming

**Make programmers sleep better!**

## Lessons

- > Concurrency does come naturally to the O-O world
- > Must revise usual modes of reasoning about programs
- > Design by Contract the key
- > A simple extension is possible
- > The mechanism can be quite general
- > We can bring concurrent programming to the same level of safety and elegance as traditional programming
- > We don't really have a choice

SCOOP is here today, try it!

[se.ethz.ch/research/scoop.html](http://se.ethz.ch/research/scoop.html)

## Status

- > All of SCOOP except duels implemented
- > Preprocessor and library available for download
- > Numerous examples available for download

[se.ethz.ch/research/scoop.html](http://se.ethz.ch/research/scoop.html)

We are very grateful to the Hasler Foundation for their support.

## Current developments & open problems

Semantic specification  
Enriched type system  
Wait on first of several events

Distribution and web services  
Support for transactions  
Deadlock prevention and detection  
Extensions for real-time  
Integration with compiler