



Concurrent O-O Programming for Real-Time Systems?

Bertrand Meyer

ETH, Zurich

Eiffel Software, Santa Barbara



Reluctance in real time field

- Image of unproven technology
- Global performance concerns
- Response-time concerns



Focus of this presentation

- Concurrency mechanism
- SCOOP:
 - Simple Concurrent Object-Oriented Programming
- Work with: Volkan Arslan, Piotr Nienaltowski, ETH



Concurrency & multithreading

- Everyone wants to do it
- Many are doing it
- Those who are doing it are not doing it very well



Impedance mismatch

- O-O: high-level abstraction mechanisms
- Concurrency: semaphores, locks, suspend, manual exclusion, sharing...



Why O-O?

- | | |
|---------------------------------------|------------------------------------|
| ▪ Structuring concept:
the class | Computation concept:
the object |
| ▪ Module-type fusion | ▪ Modeling power |
| ▪ Information hiding | ▪ Dynamic allocation |
| ▪ Multiple inheritance | ▪ Automatic memory
management |
| ▪ Genericity | |
| ▪ Polymorphism and
dynamic binding | |
| ▪ Contracts | |

X.r (a)



Reasoning about objects

$$\{ \text{Pre}_r \quad \} \quad \text{body}_r \quad \{ \text{Post}_r \quad \}$$

$$\{ \text{Pre}_r' \} \text{ x.r (a) } \{ \text{Post}_r' \}$$


Reasoning about objects

$$\{ \text{Pre}_r \text{ and INV} \} \quad \text{body}_r \quad \{ \text{Post}_r \text{ and INV} \}$$

$$\{ \text{Pre}_r' \} \text{ x.r (a) } \{ \text{Post}_r' \}$$

Reasoning about objects

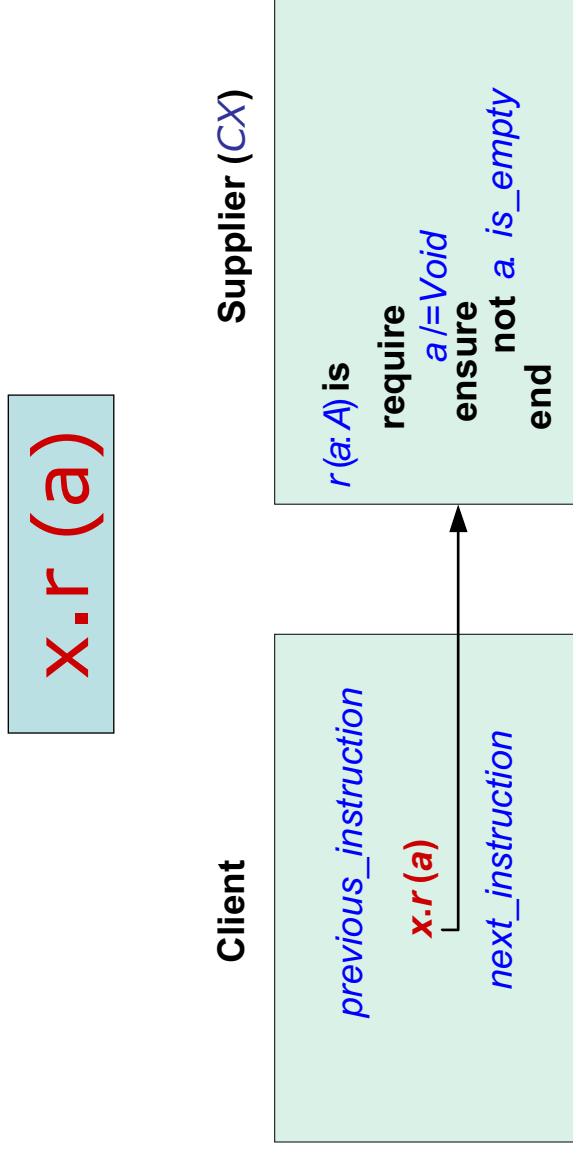
Only n proofs if n exported routines!

$$\{ \text{Pre}_r \textbf{ and INV} \} \text{ body}_r \{ \text{Post}_r \textbf{ and INV} \}$$

$$\{ \text{Pre}_r \}' \mathbf{x.r(a)} \{ \text{Post}_r \}'$$

Feature call

$x: CX$



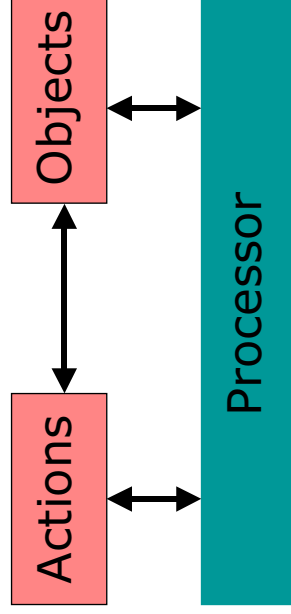


Object-oriented computation

To perform a computation is

- to apply certain **actions**
- to certain **objects**
- using certain **processors**

X.r (a)



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



Mutual exclusion on calls to same object

Only n proofs if n exported routines!

$\{Pre_r \text{ and } INV\}$ $body_r$ $\{Post_r \text{ and } INV\}$

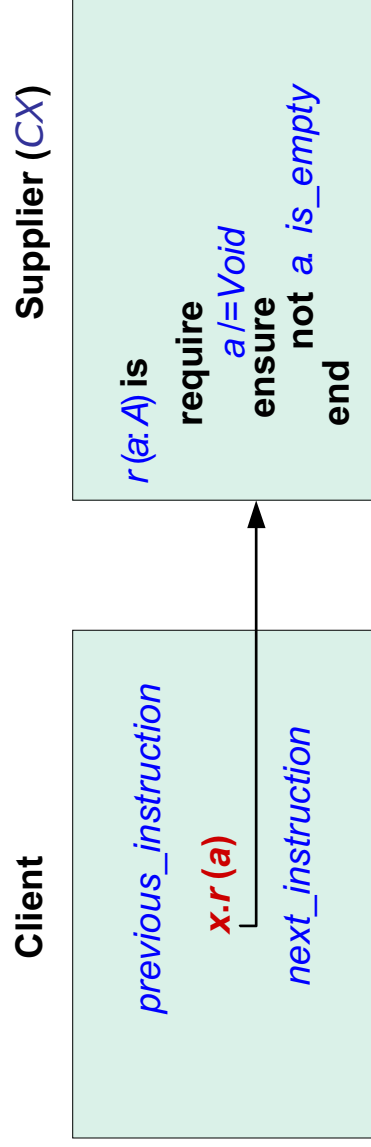
$\{Pre_r\}$ $x.r(a)$ $\{Post_r\}$



Feature call: sequential

$x.r(a)$

$x: CX$



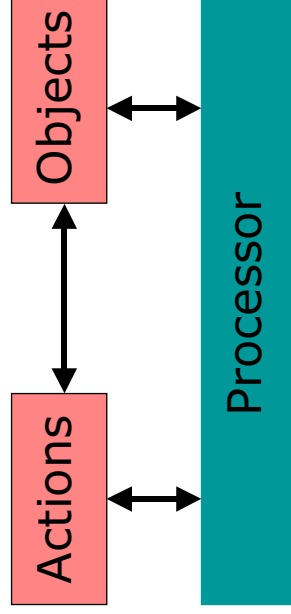


Object-oriented computation

To perform a computation is

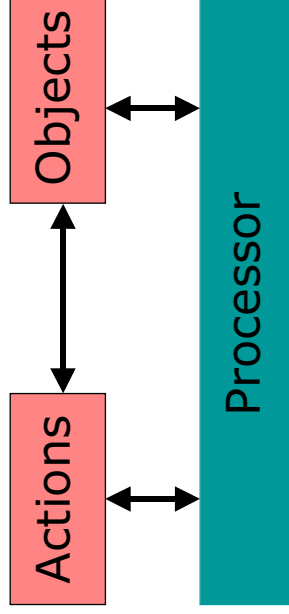
- to apply certain **actions**
- to certain **objects**
- using certain **processors**

x.r (a)



Object-oriented computation

Each object **handled** by a specific processor.

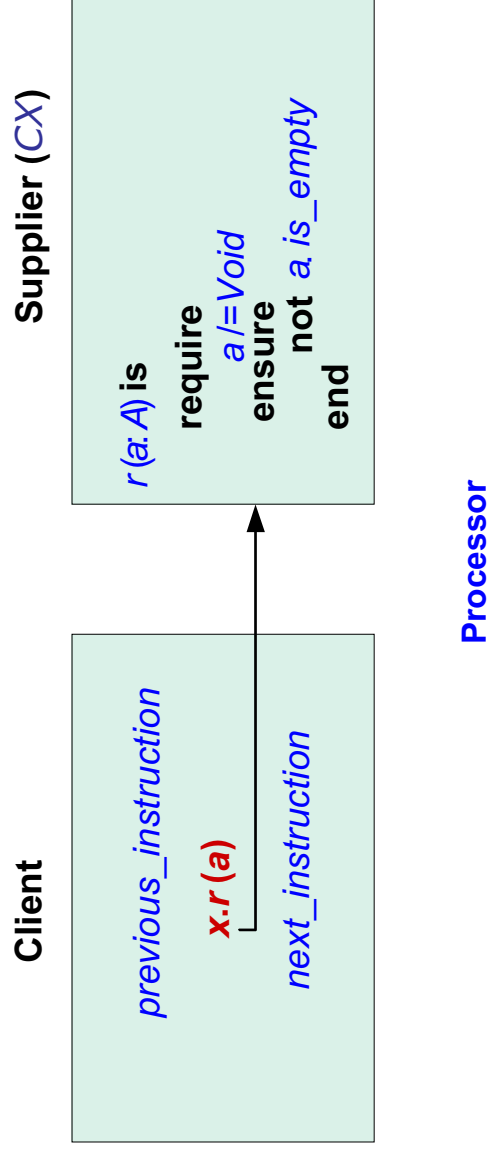




Feature call: sequential

x.r (a)

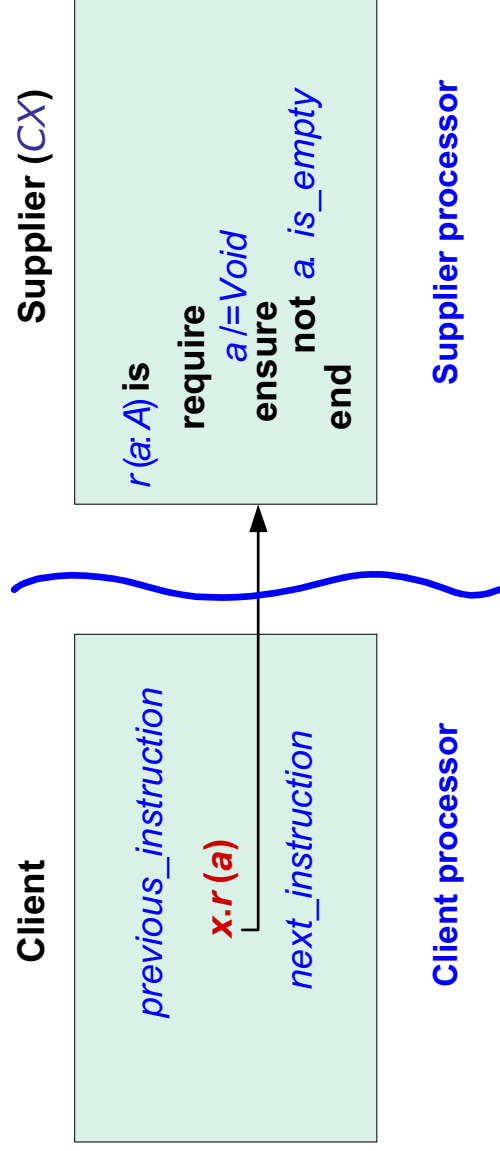
x: CX



Feature call: asynchronous

x.r (a)

x: CX





The fundamental difference

To wait or not to wait:

- If same processor, **synchronous**
- If different processor, **asynchronous**

Difference must be captured by syntax:

- x: CX
- x: **separate** CX



What does "separate" mean?

- Does not specify processor
- Simply indicates that it's "elsewhere"



If no access control

x: **separate CX**

...

x.r (a)



y := x.f



If no access control

x: **separate CX**

...

my_stack.push (a)



y := my_stack.top



Access control policy

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

```
push (b: separate STACK [T]; value: T) is  
  -- Store value into b.  
do  
  b.push (value)  
end
```



Access control policy

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

```
push (b: separate BUFFER [T]; value: T) is  
  -- Store value into b.  
do  
  b.put (value)  
end
```

- To use separate object:
 my_buffer: **separate** BUFFER [INTEGER]
 create *my_buffer*
 store (*my_buffer*, 10)



Mutual exclusion on calls to same object

Only n proofs if n exported routines!

$\{\text{Pre}_r \text{ and INV}\}$ body _{r} $\{\text{Post}_r \text{ and INV}\}$

$\{\text{Pre}_r\}$ x.r (a) $\{\text{Post}_r\}$



Mutual exclusion rule

- Numerous optimizations possible
- No need to retain object longer than needed



Design by Contract

Supplier:

```
store (b: BUFFER [T]; value: T) is
  -- Store value into b.
  require
    not b.is_full
    value > 0
  do
    b.put (value)
  ensure
    not b.is_empty
  end
  ...
```

Client:

```
if not my_buffer.is_full
  then
    store (my_buffer, x)
  end
```



Contracts under concurrency?

Supplier:

```
store (b: BUFFER [T]; value: T) is
  -- Store value into b.
  require
    not b.is_full
    value > 0
  do
    b.put (value)
  ensure
    not b.is_empty
  end
  ...
```

Client:

```
if not my_buffer.is_full
  then
    store (my_buffer, x)
  end
```





What happens with preconditions?

- Precondition on separate target becomes **wait condition** (instead of correctness condition)
- This becomes the basic synchronization mechanism



Resynchronization

- No special 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 by necessity

- No special 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
```



Wait by necessity

```
x.f  
x.g (a)  
y.f  
...  
value := x.some_query  
...  
if value > 10 then ... end     ← Wait here
```



Interrupts?

Can we snatch shared object from its current holder?

- Execute *holder.r (b)* where *b* is **separate**
- Another object executes *challenger.s (b)*
- Normally, *challenger* would wait
- What if *challenger* is impatient?



The duel mechanism

Library features

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



Extending duels with priorities

- `holder.set_priority (50)`
- `challenger.set_priority (100)`
- `holder.yield`
- `challenger.immediate_service`

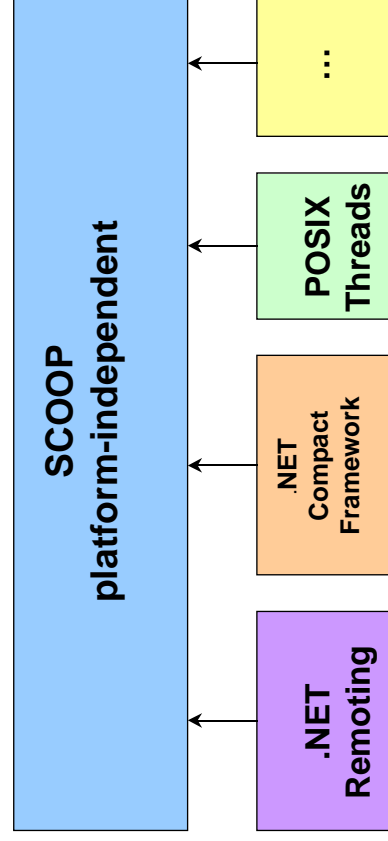
If the priority of the *challenger* is bigger than the priority of the *holder* (`challenger.priority > holder.priority`):

- *holder* will get an exception
- *challenger* will be served.



Two-level architecture of SCOOP

- Adaptable to many environments
- .NET remoting is current platform





Mapping processors to resources

- Location of processors need not be specified at compile time
- On the fly specification with Concurrency Control File (CCF)



Concurrency Control File

```
creation  
system  
    "wolfgang" (4): "c:\appl1.exe"  
    "amadeus" (2): "http://greatsite.com/prog.dll"  
end  
external  
    database_handler: "Lodron" port 9000  
    ATM_handler:    "Colloredo" port 8001  
end  
default  
    port: 8001; instance: 10  
end
```



Example: Bounded buffer usage

Usage of bounded buffers

```
buff: BUFFER_ACCESS [MESSAGE]
my_buffer: BOUNDED_BUFFER [MESSAGE]

create my_buffer
create buff.make (my_buffer)

buff.put (my_buffer, my_message)
...
buff.put (my_buffer, her_message)
...
my_query := buff.item (my_buffer)
```



Example: Dining philosophers

```
class PHILOSOPHER inherit
```

```
...
```

```
feature {BUTLER}
```

```
step is
```

```
-- Perform tasks.
```

```
do
```

```
  think; eat (left, right)
```

```
end
```

```
eat (l, r: separate FORK) is
```

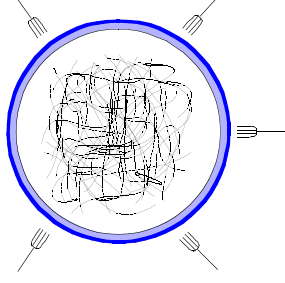
```
-- Eat, having grabbed l and r.
```

```
do
```

```
...
```

```
end
```

```
end
```





Class *PROCESS*

indexing

description: "The most general notion of process"

deferred class *PROCESS*

feature -- Status report

over: **BOOLEAN is**

-- Must execution terminate now?

deferred end

feature -- Basic operations

setup is

-- Prepare to execute process operations (default: nothing).

do end

step is

-- Execute basic process operations.

deferred end



Class *PROCESS* (cont.)

wrapup is

-- Execute termination operations (default: nothing).

do end

feature -- Process behavior

live is

-- Perform process lifecycle.

do

from *setup* **until** *over loop*

step

end

wrapup

end

end



Other examples

- Watchdog: use duels
- Elevator systems
- Others in the book



Implementation

- SCOOPLI for .NET and others
 - Library implementation
 - Uses *Remoting* and *Threading* capabilities of .NET
 - Uses *Threading* capabilities of the .NET CF on the RTOS Windows CE .NET (and/or other RTOS: VxWorks, etc.)
- XCOOPLI
 - Platform for experimenting with extensions



Extensions

- Real-time: timed assertions, priorities for duels
- Persistence
- Deadlock detection and prevention



The scope of SCOOP

Search for generality and simplicity

- Full concurrency
- Full use of O-O
- Minimal extension to basic O-O scheme: keyword **separate** + library features
- Based on *Design by Contract*
- Multi-platforms and architecture
- Makes it much simpler to use concurrency