Concurrent Object-Oriented Programming

Bertrand Meyer, Volkan Arslan
Lecture 3: Computational model of SCOOP
Outline

- Processors
- Separate objects, separate entities
- Synchronous and asynchronous call semantics
- Mutual exclusion
- Condition synchronisation
- Wait-by-necessity
Basic idea of OO computation

To perform a computation is
- To apply certain actions
- To certain objects
- Using certain processors
Processors

- Processor: a thread of control supporting **sequential execution** of instructions on one or several objects.

- All actions on a given object are executed by its **handling processor**. **No shared access to objects!**

- We say that the object is **handled** by its processor.
  - This relationship is **fixed**, i.e. we do not consider migration of objects between processors.

- Each processor, together with all object it owns, can be seen as a sequential subsystem.

- A (concurrent) **software system** is composed of such subsystems.
P1 handles o1, o2, o3, o4
P2 handles o5, o7, o9
P3 handles o6, o8, o11, o12

<o1> denotes o1’s handler
<o1> = P1
Processors (cont.)

- Processor is an abstract concept
- Do not confuse it with a CPU!

- A processor can be implemented as:
  - Process
  - Thread
  - Web service
  - .NET AppDomain
  - ???
Feature call - synchronous

\[ x \in X \]

\[ \ldots \]

\[ x.f(a) \]

\[ \text{previous\_instruction} \]

\[ x.f(a) \]

\[ \text{next\_instruction} \]

\[ f(a : A) \]

\[ \text{require} \]

\[ a \neq \text{Void} \]

\[ \text{do} \]

\[ \ldots \]

\[ \text{end} \]
Feature call - asynchronous

\( x: \text{separate } X \)

\( \ldots \)

\( x.f(a) \)

\( \text{previous\_instruction} \)

\( x.f(a) \)

\( \text{next\_instruction} \)

\( (\text{CLIENT}) \)

\( f(a: A) \)

\( \text{require} \)

\( a \neq \text{Void} \)

\( \text{do} \)

\( \ldots \)

\( \text{end} \)

\( (X) \)
Separate objects

- Calls to non-separate objects are synchronous
- Calls to separate objects are asynchronous

QUIZ: Which objects are separate here?
Separate entities are declared with `separate` keyword

\[ x : \text{separate} \ X \]

Does a separate entity always denote a separate object?

\[ y : \text{separate} \ X \]
\[ r (x : \text{separate} \ X) \text{ is do} \]
\[ y := x \cdot y \]  -- Is \( y \) a separate entity?
\[ -- \text{Does it denote a separate object?} \]

Separate entities denote potentially separate objects
Synchronisation

- Processors are **sequential**

- Concurrency is achieved by **interplay** of several processors

- Processors need to **synchronise**

- Three forms of synchronisation in SCOOP
  - mutual exclusion
  - condition synchronisation
  - wait-by-necessity
If no mutual exclusion

- Programmer writes:

```plaintext
my_stack: separate STACK [INTEGER]
```

...  

```plaintext
my_stack.push (5)  
y := my_stack.top  
```

What could have happened here?

--- Are we sure that \( y = 5 \) ?

We need a critical section to avoid atomicity violations.
Problematic scenario

P1 and P3 execute similar code:

-- P1
my_stack.push (5)
y := my_stack.top

-- P3
my_stack.push (100)
y := my_stack.top
Mutual exclusion in SCOOP

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

  \[
  \text{push\_and\_retrieve (s: separate STACK [INTEGER]; value: INTEGER)}
  \]
  \[
  \quad \text{-- Push value on top of s then retrieve top of s}
  \]
  \[
  \quad \text{-- and assign it to y.}
  \]

  \[
  \begin{aligned}
  \text{do} & \quad \text{s.push (value)} \\
  \text{end} & \quad \text{y := s.top}
  \end{aligned}
  \]

  \[
  \begin{aligned}
  \text{No other processor can access s in the meantime}
  \end{aligned}
  \]

\[
\text{my\_stack: separate STACK [INTEGER]}
\]
\[
\ldots
\]
\[
\text{push\_and\_retrieve (my\_stack, 5) -- Now we are we sure that y=5}
\]

- Body (do ... end) of enclosing routine is a critical section with respect to its separate formal arguments.
Separate argument rule

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

Separate call: $a.f(\ldots)$ where $a$ is a separate entity
A routine call with separate arguments will execute when all corresponding processors are available

and hold them exclusively for the duration of the routine
Synchronisation

- Three forms of synchronisation in SCOOP
  - mutual exclusion
  - condition synchronisation
  - wait-by-necessity
Condition synchronisation

- Very often client only wants to execute certain feature if some condition (guard) is true:

```plaintext
store (buffer: separate BOUNDED_BUFFER [INTEGER]; value: INTEGER) is
  -- Store value into buffer.
  require
    buffer_not_full: not buffer.is_full
  do
    buffer.put (value)
  end

my_buffer: separate BOUNDED_BUFFER [INTEGER]
...
store (my_buffer, 5)
```

Hey, it’s a precondition, not a guard!

How should it work?
store (buffer: BUFFER [INTEGER]; value: INTEGER)

is

-- Store value into buffer.

require
  buffer_not_full: not buffer.is_full
  value > 0

do
  buffer.put (value)

ensure
  buffer_not_empty: not buffer.is_empty

end

...

store (my_buffer, 5)
store (buffer: separate BUFFER [INTEGER]; value: INTEGER)
is
  -- Store value into buffer.
require
  buffer_not_full: not buffer.is_full
  value > 0
do
  buffer.put (value)
ensure
  buffer_not_empty: not buffer.is_empty
end

... 
store (my_buffer, 5)
Why new semantics?

- Preconditions are obligations that client has to satisfy before the call

\[ \{ \text{Pre}_r \} \text{ call r } \{ \text{Post}_r \} \]

- Easy peasy:

```java
if not my_buffer.is_full and then 5 > 0 then
    store (my_buffer, 5)
end
```

I know that precondition holds before the call!
A routine call with separate arguments will execute when all corresponding processors are available and preconditions are satisfied and hold the objects exclusively for the duration of the routine.
Resynchronising clients and suppliers

...  \( x.f \)  \( x.g(a) \)  \( y.f \)  ...

\[ \text{Pc} \]

\[ \text{Px} \]

\[ \text{Py} \]

\[ x.f \]

\[ x.g(a) \]

\[ y.f \]
Synchronisation

- Three forms of synchronisation in SCOOP
  - mutual exclusion
  - condition synchronisation
  - wait-by-necessity
Wait by necessity

- No explicit mechanism for resynchronisation after separate call.

- Client will only wait when it needs to:
  
  \[ x.f \]
  \[ x.g(a) \]
  \[ y.f \]
  ...

  \[ value := x.some_query \]

- This is called **wait-by-necessity**
Do we really need to wait?

- Can we do better than that?
  
  \[
  x.f \\
  x.g(a) \\
  y.f \\
  ... \\
  value := x.some_query \\
  x.f \\
  y.f \\
  z := value \\
  value := value + 1
  \]

- Does not change the basic SCOOP model
- Consider it to be an optimisation
A problem

...  
$x.f$

$x.g(y)$

...  
$y.f$

g(y: separate Y) is do

$\ldots$

end
Lock passing

- Original SCOOP approach:
  - Make x wait until y becomes available
  - “Business Card principle” for dealing with tricky cases
  - Not flexible

- Lock passing approach:
  - Let x get exclusive access on y immediately
  - “Pass the lock on y”
  - But: client that passes the lock has to wait
  - In fact, client can pass all the locks
  - You can still implement previous scenario
Lock passing

\[ r(x: \text{separate } X; y: \text{separate } Y) \text{ is} \]
\[ \text{do} \]
\[ x.f \]
\[ x.g(y) \quad \text{-- Pass your locks to } x \text{ and wait for } x \text{ to finish.} \]
\[ y.f \]
\[ ... \]
\[ \text{value} := x.\text{some_query} \]
\[ \text{end} \]

Both calls are synchronous!
Summary: computational model

- Software system is composed of several **processors**
- Processors are **sequential**; concurrency is achieved through their interplay
- Separate entity denotes a **potentially separate object**
- Calls to non-separate objects are **synchronous**
- Calls to separate objects are **asynchronous**
Summary: computational model

- Mutual exclusion
  - Locking through argument passing
  - Routine body is critical section

- Condition synchronisation
  - wait-semantics for preconditions

- Re-synchronisation of client and supplier:
  - wait-by-necessity

- Lock passing through argument passing
A routine call with separate arguments will execute when all corresponding objects are available and wait-conditions are satisfied and hold the objects exclusively for the duration of the routine.