Object-Oriented Software Construction

Bertrand Meyer

Lecture 19: SCOOP
Simple Concurrent Object-Oriented Programming

Piotr Nienaltowski
What we’re going to see today

- Some history
- Back to the roots: the essence of OO
- SCOOP: computational model
- Synchronisation
- Advanced mechanisms
- How does it fit into the OO framework?
The need for concurrency: where

- Operating systems
  - We already forgot the “batch processing” systems from the 60s
  - Users want multi-tasking

- Distributed applications

- Modern GUIs

- Many applications only make sense in the presence of concurrency

- We want to squeeze maximum power from our computers
  - Use additional CPUs
The need for concurrency: how

- Two basic kinds of concurrency
  - Cooperative
  - Competitive

- Cooperative
  - several tasks want to achieve a common goal
  - synchronisation is necessary to achieve the goal
  - example: producer-consumer

- Competitive
  - several task access resources to do their own job
  - no common goal; tasks often don’t know about each other
  - synchronisation is necessary to avoid conflicts
  - example: flight booking system
The need for concurrency: what

- Two kinds of properties of interest
  - Safety (nothing bad happens)
  - Liveness (something good eventually happens)

- Data races
- Deadlocks
- Livelocks
A bit of history

- Mutex
- Semaphore (60s - Dijkstra)
- Conditional Critical Region (1968, Hoare)
- Monitor (1972, Brinch-Hansen, Hoare)
- Rendez-vous (80s, Ada), active objects
- Simple Concurrent Object-Oriented Programming
- Described in CACM, 1993
- OOSC, 2nd edition, 1997
- Prototype implementation at Eiffel Software, 1995
- Prototypes by others
- Now being done for good at ETH (support from SNF, Hasler Foundation, Microsoft ROTOR)
Basic idea of OO computation

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

\[ x.f(a) \]
Computational model of SCOOP

- 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 memory!!!*

- We say that an object is **owned** by its handling processor
  - this ownership relation 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.
Software system

P1

- o1
- o2
- o3
- o4

P2

- o5
- o7
- o9

P3

- o6
- o8
- o11
- o12

P1 handles o1, o2, o3, o4
P2 handles o5, o7, o9
P3 handles o6, o8, o11, o12

<o1> denotes o1’s owner
<o1> = P1
Processors

- 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

\[ x : X \]
\[
... \\
x.f(a) \\
\]

**P1**

**o1**

previous\_instruction

\[ x.f(a) \]

next\_instruction

**o2**

\[ f(a : A) \]

require

\[ a /= Void \]

do

\[
... \\
end \\
\]

**(X)**

**(CLIENT)**
Feature call - asynchronous

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

... 

\[ x.f(a) \]
Separate objects

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

QUIZ: Which objects are separate?
Separate entities

- Separate entities are declared with `separate` keyword
  \[ x: \textit{separate} \ X \]

- Does a separate entity always denote a separate object?
  \[ x, y: \textit{separate} \ X \]
  ...  
  \[ y := x.y \quad -- \text{Is } y \text{ a separate entity?} \]
  \[ -- \text{Does it denote a separate object?} \]

- Separate entities denote \textit{potentially separate} objects
Synchronisation

- Processors are **sequential**

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

- Processors need to **synchronise**

- Two forms of synchronisation in SCOOP
  - mutual exclusion
  - condition synchronisation
If no mutual exclusion

- Programmer writes:

  \[
  \textit{my\_stack}: \textbf{separate} \textit{STACK [INTEGER]} \\
  \]

  ... \\

  \[
  \begin{align*}
  &\text{\textcolor{red}{my\_stack.push (5)}} & \text{What could have happened here?} \\
  &y := \textit{my\_stack.top} & \text{-- Are we sure that } y = 5 \, ? \\
  \end{align*}
  \]

  We need a \textbf{critical section} to avoid data races.
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:

```haskell
push_and_retrieve (s: separate STACK [INTEGER];
    value: INTEGER) is
    -- Push `value' on top of `s' then retrieve top of `s'
    -- and assign it to `y'.

    do
        s.push (value)
        y := s.top
    end
```

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

```haskell
my_stack: separate STACK [INTEGER]
...
push_and_retrieve (my_stack, 5) -- Now we are we sure that y=5
```
The target of a separate call must be a formal argument of the enclosing routine

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

and hold them exclusively for the duration of the routine
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?
Contracts strike back 😊

\[
\text{store (buffer: BUFFER [INTEGER]; value: INTEGER)} \\
\text{is} \\
\text{-- Store `value` into `buffer`.} \\
\text{require} \\
\text{buffer\_not\_full: } \text{not buffer.is\_full} \\
\text{value > 0} \\
\text{do} \\
\text{buffer.put (value)} \\
\text{ensure} \\
\text{buffer\_not\_empty: } \text{not buffer.is\_empty} \\
\text{end} \\
\]
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, 10)
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:

```plaintext
if \text{precondition}
store \text{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 objects are available and wait-conditions are satisfied and hold the objects exclusively for the duration of the routine
Re-synchronising clients and suppliers

... Pc

$\text{x.f}$ \rightarrow \text{Px}

$\text{x.g(a)}$ \rightarrow \text{Py}

$\text{y.f}$

...
Wait by necessity

- No special mechanism for client to re-synchronise with supplier after separate call.
- Client will only wait when it needs to:
  
  \[
  x.f \\
  x.g (a) \\
  y.f \\
  \ldots \\
  \text{value := } x.\text{some\_query}
  \]

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

- Can we do better than that?
  
  \[ x.f \]
  
  \[ x.g(a) \]
  
  \[ y.f \]
  
  ...
  
  \[ \text{value} := x.\text{some}_{\text{query}} \]
  
  \[ x.f \]
  
  \[ y.f \]
  
  \[ z := \text{value} \]
  
  \[ \text{value} := \text{value} + 1 \]

- Does not change the basic SCOOP model
- Consider it to be an optimisation
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: synchronisation

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

- Condition synchronisation
  - preconditions

- Re-synchronisation of client and supplier:
  - wait-by-necessity
The target of a separate call must be a formal argument of the enclosing routine

Separate call: $a.f(...)$ where $a$ is a separate entity
Summary: wait rule

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
Now that we know (almost) everything, let’s see a short example.
Is that enough to prevent data races?

- **Data race occurs when two or more clients concurrently apply some feature on the same supplier.**

- Data races could be caused by so-called **traitors**, i.e. non-separate entities that denote separate objects.
  - **Kill ’em all!**
--- in class C (client)
x: separate X
a: A
...
\( r \) \( (an_x: \text{separate } X) \) is
  do
    a := an_x.a
  end
...
\( r \) \( (x) \)
\( a.f \)

--- supplier

class X
feature
  a: A
end

Is this call valid?

TRAITOR! TRAITOR!

And this one?
Consistency rules – first attempt

- Four consistency rules
- Should prevent data races
  - eliminate traitors
- Written in English
- Easy to understand by programmers
**SCOOP rules – first attempt**

---

**Separateness consistency rule (1)**

If the source of an attachment (assignment instruction or argument passing) is separate, its target entity must be separate too.

\[
\begin{align*}
  r \ (\textbf{buffer}: \ & \textbf{separate} \ BUFFER \ [X]; \ x: \ X) \ \textbf{is} \\
  & \textbf{local} \\
  & b1: \ \textbf{separate} \ BUFFER \ [X] \\
  & b2: \ BUFFER \ [X] \\
  & x2: \ \textbf{separate} \ X \\
  \ & \textbf{do} \\
  & b1 := \textbf{buffer} \ -- \ \textbf{valid} \\
  & b2 := b1 \quad -- \ \textbf{invalid} \\
  & r \ (b1, \ x2) \quad -- \ \textbf{invalid} \\
  \ & \textbf{end}
\end{align*}
\]

---

*Chair of Software Engineering*
**Separateness consistency rule (2)**

If an actual argument of a separate call is of a reference type, the corresponding formal argument must be declared as separate.

```
store (buffer: separate BUFFER [X]; x: X) is
  do
    buffer.put (x)
  end

-- in class BUFFER [G]
put (element: separate G) is
  ...
```
Separateness consistency rule (3)
If the source of an attachment is the result of a separate call to a function returning a reference type, the target must be declared as separate.

```plaintext
consume_element (buffer: separate BUFFER [X]) is
    local
        element: separate X
    do
        element := buffer.item
        ...
    end
    -- in class BUFFER [G]
    item: G is
        ...
```

Chair of Software Engineering
Separateness consistency rule (4)
If an actual argument of a separate call is of an expanded type, its base class may not include, directly or indirectly, any non-separate attribute of a reference type.

store (buffer: separate BUFFER [X]; x: X) is
  do
    buffer.put (x) -- X must be “fully expanded”
  end

-- in class BUFFER [G]
put (element: G) is -- G is not declared as separate anymore
  . . .
Consistency rules – second attempt

- What can a type system do better than a set of informal rules?
- Prevent data races
  - static (compile-time) checks
- Integrate expanded types and agents with SCOOP
- Ownership-like types
  - Eiffel types augmented with owner tags
  - inspired by Peter Mueller’s work on applet isolation in JavaCard
- **Subtyping rule** replaces all previous consistency rules
Type system

Let TypeId denote the set of declared type identifiers of a given Eiffel program. We define the set of tagged types for a given class as

$$TaggedType = OwnerId \times TypeId$$

where OwnerId is a set of owner tags declared in the given class. Each class implicitly declares two owner tags: • (current processor) and ⊥ (undefined).

The subtype relation $\prec$ on tagged types is the smallest reflexive, transitive relation satisfying the following axioms, where $\alpha$ is a tag, $S, T \in TypeId$, and $\prec_{Eiffel}$ denotes the subtype relation on TypeId:

$$(\alpha, T) \prec (\alpha, S) \iff T \prec_{Eiffel} S$$

$$(\alpha, T) \prec (\perp, T)$$
**False traitors**

meet_someone_elses_friend (person: separate PERSON) is

local

    a_friend: PERSON

do

    a_friend := person.friend -- Invalid assignment.
    visit (a_friend)

end

*Chair of Software Engineering*
**Handling false traitors**

```
meet_someone_elses_friend (person: separate PERSON) is
    local
        a_friend: PERSON
    do
        a_friend := person.friend -- Valid assignment attempt.
        if a_friend /= void then visit (a_friend) end
    end
```
Interrupts?

Can we snatch a shared object from its current holder?

Execute \textit{holder.r} (\textit{b}) where \textit{b} is separate

Another object executes \textit{challenger.s} (\textit{b})

Normally, \textit{challenger} would wait

What if challenger is impatient?
### Library features

<table>
<thead>
<tr>
<th></th>
<th>challenger</th>
<th>normal_service</th>
<th>immediate_service</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>holder</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>retain</strong></td>
<td>challenger waits</td>
<td>exception in challenger</td>
<td></td>
</tr>
<tr>
<td><strong>yield</strong></td>
<td>challenger waits</td>
<td>exception in holder; serve challenger</td>
<td></td>
</tr>
</tbody>
</table>
How does it fit within the OO?

- Basic mechanism: feature call
- Design by Contract
  - Generalised semantics for preconditions, postconditions, and invariants
- Inheritance
  - No particular restrictions, usual rules apply
  - Most inheritance anomalies eliminated!
- Genericity: full support
- Agents: almost full support
  - Unclear semantics of agents with open target
Genericity

my_array: ARRAY [X]

my_array: separate ARRAY [X]

my_array: ARRAY [separate X]

my_array: separate ARRAY [separate X]
Problem: agents

\( a: \ \text{PROCEDURE} \ [\text{ANY, TUPLE}] \)

\[
\text{store} \ (\text{buffer: separate} \ \text{BUFFER} \ [X]; \ x: \ X) \ \text{is}
\]

\[
\begin{align*}
& \text{do} \\
& \quad \text{buffer.put} \ (x) \\
& \quad a := \text{agent} \ \text{buffer.put} \ (?) \quad -- \text{Valid!} \\
& \quad \text{fishy\_call} \ (x) \\
& \end{align*}
\]

\[
\text{fishy\_call} \ (x: \ X) \ \text{is}
\]

\[
\begin{align*}
& \text{do} \\
& \quad a.\text{call} \ ([x]) \quad -- \text{Valid! But } a \text{ is a traitor!} \\
& \end{align*}
\]
Solution: separate agents

a: separate PROCEDURE [ANY, TUPLE]

store (buffer: separate BUFFER [X]; x: X) is
    do
        buffer.put (x)
        a := agent buffer.put (?)     -- separate agent
        fishy_call (x)
    end

fishy_call (x: X) is
    do
        a.call ([x])     -- Invalid: a is not a formal argument.
    end
Separate agents

\[a: \text{separate } \textit{PROCEDURE } [\text{ANY, TUPLE}]\]

\[\text{store } (\text{buffer: separate } \textit{BUFFER } [X]; \ x: \ X \ ) \ \textbf{is}\]

\[\text{do}\]

\[\text{buffer.put } (x)\]

\[a := \textbf{agent} \text{ buffer.put } (?)\]

\[\text{not_so_fishy_call } (a, x)\]

\[\text{end}\]

\[\text{not_so_fishy_call } (\textbf{an_agent: separate } \textit{PROCEDURE } [\text{ANY, TUPLE}]; \ x: \ X \ ) \ \textbf{is}\]

\[\text{do}\]

\[\text{an_agent.call } ([x]) \quad \text{-- Valid: } a \text{ is a formal argument.}\]

\[\text{end}\]

\[\text{different semantics for argument passing: lock an_agent.target}\]
Damn! I was very fast and I need to fill the remaining time with something interesting.

Do you want to see another example or go home?
That’s all, folks!

Thanks!

And don’t forget to SCOOP it up!