Object-Oriented Software Construction

Bertrand Meyer

Lecture 19: SCOOP
Simple Concurrent Object-Oriented Programming

Piotr Nienaltowski

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

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: 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

SCOOP

- 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)

A bit of history

- Mutex
- Semaphore (60s - Dijkstra)
- Conditional Critical Region (1968, Hoare)
- Monitor (1972, Brinch-Hansen, Hoare)
- Rendez-vous (80s, Ada), active objects

Basic idea of OO computation

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

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
  - ???

Software system

Feature call - synchronous

\( x: X \)

\[ \ldots \]

\( x.f(a) \)

\( o1 \)

\( o2 \)

previous_instruction

\( x.f(a) \)

next_instruction

\((\text{CLIENT})\)

\( (X) \)
**Feature call - asynchronous**

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

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

\[ o1 \]

\[ \text{previous instruction} \]

\[ x.f \ (a) \]

\[ \text{next instruction} \]

\[ (\text{CLIENT}) \]

\[ o2 \]

\[ f \ (a: A) \]

\[ \text{require} \ a 
\begin{align*}
\text{do} \\
\text{end} \\
\end{align*} \]

\[ \text{Void} \]

\[ (X) \]

\[ \text{P1} \]

\[ \text{P2} \]

**Separate entities**

- Separate entities are declared with `separate` keyword

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

- Does a separate entity always denote a separate object?

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

\[ y := x.y \quad \text{-- Is y a separate entity?} \]

\[ \text{-- Does it denote a separate object?} \]

- Separate entities denote potentially separate objects

**Separate objects**

- Calls to non-separate objects are synchronous

- Calls to separate objects are asynchronous

**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

**QUIZ:** Which objects are separate?
If no mutual exclusion

- Programmer writes:

  
  \( \text{my\_stack: separate STACK [INTEGER]} \)
  
  ...
  
  \( \text{my\_stack.push (5)} \)
  
  \( y := \text{my\_stack.top} \quad -- \text{Are we sure that } y = 5 ? \)
  
  We need a critical section to avoid data races.

Mutual exclusion in SCOOP

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

  
  \( \text{push\_and\_retrieve (s: separate STACK [INTEGER]);} \)
  
  \( \text{value: INTEGER) is} \)
  
  \( \text{-- Push `value` on top of `s` then retrieve top of `s`} \)
  
  \( \text{-- and assign it to `y`.} \)
  
  \( \begin{array}{l}
  \text{do} \\
  \text{s.push (value)} \\
  \text{end}
  \end{array} \)
  
  \( y := s.top \quad \text{No other processor can access s in the meantime!} \)

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

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

Problematic scenario

P1 and P3 execute similar code:

- P1

  \( \text{my\_stack.push (5)} \)
  
  \( y := \text{my\_stack.top} \)

- P3

  \( \text{my\_stack.push (100)} \)
  
  \( y := \text{my\_stack.top} \)

Separate argument rule

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
**Wait rule**

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

and hold them exclusively for the duration of the routine

---

**Contracts strike back 😊**

```plaintext
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, 10)
```

---

**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
  Hey, it's a precondition, not a guard!
  How should it work?
do
  buffer.put (value)
end

my_buffer: separate BOUNDED_BUFFER [INTEGER]

... store (my_buffer, 5)
```

---

**From preconditions to wait-conditions.**

```plaintext
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)
```

---

On separate target, precondition becomes wait condition
Why new semantics?

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

\{Pre,\} call r \{Post,\}

- Easy peasy:

  \textbf{if} \texttt{precondition} \textbf{store} \texttt{then}
  \texttt{store (my_buffer, 5) \textbf{end}}

I know that precondition holds before the call!

---

Re-synchronising clients and suppliers

\[ ... \]
\[ x.f \rightarrow x.f \]
\[ x.g(a) \rightarrow x.g(a) \]
\[ y.f \rightarrow y.f \]
\[ ... \]

---

Wait rule revisited

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

---

Wait by necessity

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

- This is called \textbf{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

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**

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: separate argument rule

**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

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 traits, i.e. non-separate entities that denote separate objects.
  - Kill ‘em all!

Example: bounded buffer

Now that we know (almost) everything, let’s see a short example

Traits

```plaintext
-- in class C (client)
x: separate X
a: A
...
r (an_x: separate X) is end
do
  a := an_x.a
end
...
```

```
-- supplier
class X
feature
  a: A
```

Is this call valid?
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 (2)**
If an actual argument of a separate call is of a reference type, the corresponding formal argument must be declared as separate.

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

-- in class BUFFER [G]
put (element: separate G) is
  . . .
```

---

**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.

```scala
r (buffer: separate BUFFER [X]; x: X) is
  local
    b1: separate BUFFER [X]
    b2: BUFFER [X]
    x2: separate X
  do
    b1 := buffer -- valid
    b2 := b1 -- invalid
    r (b1, x2) -- invalid
  end
```

---

**SCOOP rules – first attempt**

**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.

```scala
consume_element (buffer: separate BUFFER [X]) is
  local
    element: separate X
  do
    element := buffer.item
  end

-- in class BUFFER [G]
item: G is
  . . .
```

---
**SCOOP rules – first attempt**

*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.

```plaintext
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 *Typeld* denote the set of declared type identifiers of a given Eiffel program. We define the set of tagged types for a given class as

\[
\text{TaggedType} = \text{OwnerId} \times \text{Typeld}
\]

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

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

\[
\begin{align*}
(\alpha, T) \prec (\alpha, S) &\iff T \prec_{\text{Eiffel}} S \\
(\alpha, T) \prec (\perp, T) &\iff \text{false}
\end{align*}
\]

**False traitors**

```
Ueli
brother
friend

Moritz
brother
friend

Urs
```

```
meet_someone_elses_friend (person: separate PERSON) is
  local
    a_friend: PERSON
  do
    a_friend := person.friend  -- Invalid assignment.
    visit (a_friend)
  end
```

```
brother
friend
brother
friend
```
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
```

Duels

<table>
<thead>
<tr>
<th></th>
<th>challenger</th>
<th>normal_service</th>
<th>immediate_service</th>
</tr>
</thead>
<tbody>
<tr>
<td>holder</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>retain</td>
<td>challenger waits</td>
<td>exception in challenger</td>
<td></td>
</tr>
<tr>
<td>yield</td>
<td>challenger waits</td>
<td>exception in holder; serve challenger</td>
<td></td>
</tr>
</tbody>
</table>

Interrupts?

Can we snatch a 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?

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: PROCEDURE (ANY, TUPLE)

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

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

Separate agents

a: separate PROCEDURE (ANY, TUPLE)

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

not_so_fishy_call (an_agent: separate PROCEDURE (ANY, TUPLE); x: X) is
  do
    an_agent.call ([x]) -- Valid: a is a formal argument.
  end

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?

---

Thanks!

And don’t forget to SCOOP it up!