SCOOP

Simple Concurrent Object-Oriented Programming
SCOOP: lectures


- Lecture 2 (17 May): validity rules, type system for SCOOP, resources.


- Lecture 5 (14 June): deadlocks and how to prevent them, future developments in SCOOP.


Exercise 3 (7 June): SCOOP tools. SCOOP examples.

Exercise 4 (14 June): SCOOP examples, project Q&A.

Exercise 5 (21 June): SCOOP examples, project
Lecture 1 (7):
Computational model of SCOOP
Outline

- Processors
- Separate objects, separate entities
- Synchronous and asynchronous call semantics
- Synchronisation
- Wait-by-necessity
- Lock passing
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 memory!!!

- We say that the object is owned by the 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 handles o1, o2, o3, o4
P2 handles o5, o7, o9
P3 handles o6, o8, o11, o12

<o1> denotes o1’s owner
<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

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

\[ previous\_instruction \]
\[ x.f(a) \]
\[ next\_instruction \]

\( (CLIENT) \)

\( f(a: A) \)
\[ require \]
\[ a /= Void \]
\[ do \]
\[ ... \]
\[ end \]

\( (X) \)
Feature call - asynchronous

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

\[ x.f(a) \]

\( o_1 \)

previous\_instruction

\( x.f(a) \)

next\_instruction

\( (\text{CLIENT}) \)

\( o_2 \)

\[ f(a: A) \]

require

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

do

\[ \ldots \]

end

\( (X) \)

P1

P2
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
  \( x: \text{separate} \ X \)

- Does a separate entity always denote a separate object?
  \( x, y: \text{separate} \ X \)
  ...
  \( y := x.y \) -- Is \( y \) a separate entity?
  -- 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

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

- Programmer writes:

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

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

We need a \texttt{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:

```
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)
  end
  y := s.top

No other processor can access s in the meantime!
```

- `my_stack: separate STACK [INTEGER]
...
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: $a.f(...)$ 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?
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)
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

\[
\{Pre_r\} \textbf{call} r \{Post_r\}
\]

- Easy peasy:

```plaintext
if \text{precondition}_{store} \text{ then }
\text{store (my_buffer, 5)}
\text{ 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

x.f → x.f → Px

x.g(a) → x.g(a) → Py

y.f → y.f → Py

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

Client will only wait when it needs to:

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

\[\text{value} := x.\text{some_query}\]

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
  ...
  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

We only need to wait here!
A problem

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

\[ x.f \rightarrow Px \]
\[ x.g(y) \rightarrow g(y: \text{separate } Y) \]
\[ g(y: \text{separate } Y) \text{ is do } y.f \]
... \[ y.f \]
... \[ y.f \]
... \[ y.f \]
end
Lock passing

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

- New approach:
  - Let x get exclusive access on y immediately
  - “Pass the lock”
  - 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

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

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-conditions

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

- Lock passing through argument passing
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
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.
Example: bounded buffer

Now that we know everything, let’s see a short example!