Concurrent Object-Oriented Programming

Bertrand Meyer, Volkan Arslan
Lecture 8:
SCOOP: Traitors, validity rules, type system
Outline

SCOOP concurrency model

- Traitors

- Validity rules – first attempt

- Type system for SCOOP
Refresher: 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
Refresher: synchronisation

- 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
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.
What SCOOP can do for us

- Beat **enemy number one** in concurrent world, i.e. atomicity violations
  - Data races
  - Illegal interleavings of calls

- Data races cannot occur in SCOOP
  - Why? See computational model ...

- Separate call rule does not protect us from bad interleaving of calls!
  - How can this happen?
-- in class C (client)

my_x: separate X

a: A

...

r (x: separate X) is

do

  a := x.a

end

...

r (my_x)

a.f

-- supplier

class X

feature

  a: A

end

Is this call valid?

Traitor! Traitor!

And this one?
-- in class C (client)

my_x: separate X

a: A

...  

r (x: separate X) is

  do

    x.f (a)

  end

...  

r (my_x)

-- supplier

class X

feature

  f (a: A) is

  do

    a.f

  end

end

Traitor! Traitor!

And this one?

Is this call valid?
Consistency rules – first attempt

- Original model defines **four consistency rules** that eliminate traitors
  - See OOSC2 chapter 30 for details

- Written in English

- Easy to understand by programmers

- Are they sound? Are they complete?
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.

\[ r \left( buffer: \text{separate} \ BUFFER [X]; \ x: \ X \right) \text{ is local} \]

\[ b1: \text{separate} \ BUFFER [X] \]
\[ b2: \ BUFFER [X] \]
\[ x2: \text{separate} \ X \]

\[ \text{do} \]
\[ b1 := buffer \quad -- \text{valid} \]
\[ b2 := b1 \quad -- \text{invalid} \]
\[ r \left( b1, x2 \right) \quad -- \text{invalid} \]

\[ \text{end} \]
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.

```plaintext
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
    ...
```
Separateness consistency rule (4)

If an actual argument or result 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
...```

---

*Chair of Software Engineering*  
Concurrent Object-Oriented Programming
Problem 1: unsoundness

$x: X -- X is expanded, see below.

consume_element (buffer: separate BUFFER [X]) is
  local
    s: STRING
  do
    x := buffer.item
    s := x.f.out   -- Valid: call on expanded object.
    s := x.g.out   -- Valid! call on separate reference.
  end

expanded class X

feature
  g: STRING
  f: INTEGER is . . .
end
Problem 2: limitations

\[ my_x : X \]  -- \( X \) is expanded, see below.
\[ my_y : \text{separate} \ Y \]
...
\[ my_y . r (my_x) \]  -- Invalid because \( my_x \) is not fully expanded.

\textbf{expanded class} \( X \)
\textbf{feature}
\begin{verbatim}
  g : STRING
  f : INTEGER is . . .
end
\end{verbatim}

\begin{verbatim}
-- in class \( Y \)
\begin{verbatim}
  r (x : X) is
     do
      ...
end
\end{verbatim}
\end{verbatim}
Problem 3: more limitations

class STRING
feature
  ...
  append alias infix "+" (other: like Current) is
    do
      ...
    end
end
-- in class X
r (l: separate LIST [STRING]) is
  do
    l.put (l.item (1) + l.item (2))
  end

Invalid!!! But it should be allowed!
Problem 4: even more limitations

\[ r \ (l: \text{separate} \ LIST\ \text{[STRING]}) \ \text{is} \]
\[
\begin{aligned}
\text{local} \\
\quad s: \text{separate} \ STRING \\
\text{do} \\
\quad s := l.\text{item} \ (1) \\
\quad l.\text{put} \ (s) \\
\text{end}
\end{aligned}
\]

Invalid!!! But it should be allowed!
Let’s make it better!

- **SCOOP rules**
  - prevent almost all traitors, +
  - are easy to understand by humans, +
  - cannot be directly used by compilers, -
  - not sound, -
  - too restrictive, -
  - no support for *agents*. -

- Can we do better?
  - Refine and *formalise* the rules!
Question

How do you know whether assignment

\[ x := y \]

-- \( x \) and \( y \) are declared as

-- \( x: \text{CLASS}_X; y: \text{CLASS}_Y \)

and argument passing

\[ r(x) \]

-- \( r \) is declared as \( r(\text{an}_x: \text{SOME}\_\text{CLASS}) \)

are valid?

Type system tells you that!
Type system for SCOOP

- Prevents all traitors
  - static (compile-time) checks

- Simplifies, refines and formalises SCOOP rules

- Integrates expanded types and agents with SCOOP
  - More about it in future lecture

- Tool for reasoning about concurrent programs
  - May serve as basis for future extensions, e.g. for deadlock prevention schemes
Three components of a type

- **Class type**
  \( x : X \)

- **Processor tag**
  \( \alpha \in \{ \bullet, T, \perp, <p>, <a.handles> \} \)

- **Attached/detachable**
  \( \gamma \in \{ !, ?, Void \} \)

\[ \Gamma \vdash x :: ( \gamma, \alpha, C ) \]
Examples

\[ x : X \quad -- \quad x :: (!, \bullet, X) \]

\[ y : \text{separate } Y \quad -- \quad y :: (!, \top, Y) \]

\[ z : ? \text{separate } Z \quad -- \quad z :: (?, \top, Z) \]

- Expanded types are attached and non-separate
  \[ i : \text{INTEGER} \quad -- \quad i :: (!, \bullet, \text{INTEGER}) \]

- **Void** is detachable
  \[ \text{Void} :: (?, \perp, \text{NONE}) \]

- **Current** is attached and non-separate
  \[ \text{Current} :: (!, \bullet, C_{\text{Current}}) \]
Examples

\[
\begin{align*}
  x & : \text{separate} <px> X \rightarrow x :: (!, px, X) \\
  y & : \text{separate} <px> Y \rightarrow y :: (!, px, Y) \\
  z & : \text{separate} <px> Z \rightarrow z :: (!, px, Z)
\end{align*}
\]

Entities \(x\), \(y\), and \(z\) represent objects handled by the same processor known as \(px\).
Subtyping rules

- Since you do not like Greek letters, we will keep it informal

- \( TT_2 \leq TT_1 \) means “\( TT_2 \) is a subtype of \( TT_1 \)”

- Conformance on class types like in Eiffel, essentially based on inheritance
  \[ D \leq_{\text{Eiffel}} C \iff (\gamma, \alpha, D) \leq (\gamma, \alpha, C) \]

- Attached \leq detachable
  \[ (!, \alpha, C) \leq (?, \alpha, C) \]

- Any processor tag \leq T
  \[ (\gamma, \alpha, C) \leq (\gamma, T, C) \]

- In particular, non-separate \leq T
  \[ (\gamma, \bullet, C) \leq (\gamma, T, C) \]

- \( \perp \leq \) any processor tag
  \[ (\gamma, \perp, C) \leq (\gamma, \alpha, C) \]
So how does it help us?

- We can rely on standard type rules
- Enriched types give us additional guarantees
- Assignment rule: source conforms to target

\[
\Gamma |- x :: TT_x, \quad \Gamma |- e :: TT_e, \quad \Gamma |- TT_e \leq TT_x
\]

[Assign] \[
\Gamma |- x := e
\]

- No need for special validity rules for separate
Examples (assignment)

\[
x : \texttt{?separate } X \quad \text{-- where } Y \text{ conforms to } X
\]
\[
y : Y
\]
\[
z : \texttt{separate } X
\]
\[
my_z : X
\]
\[
x := y
\]
\[
x := z
\]
\[
y := x
\]
\[
z := x
\]
\[
my_z := z
\]
Examples (assignment)

\[
x: \ ?\text{separate} \ X \quad -- \ x :: (? , T , X)
\]

\[
y: \ Y \quad -- \ y :: (! , \bullet , Y)
\]

\[
z: \ \text{separate} \ X \quad -- \ z :: (! , T , X)
\]

\[
my\_z: \ X \quad -- \ my\_z :: (! , \bullet , X)
\]

\[
x := y
\]

\[
x := z
\]

\[
y := x \quad -- \ \text{invalid}
\]

\[
z := x \quad -- \ \text{invalid}
\]

\[
my\_z := z \quad -- \ \text{invalid}
\]