Concurrent Object-Oriented Programming

Bertrand Meyer, Volkan Arslan
Lecture 11: SCOOP: Flexible locking and genericity
Outline

SCOOP concurrency model

- Flexible locking
  - Eliminating unnecessary locks
  - Lock passing

- Genericity
Too much locking

\[ r \left( x: \text{separate } X; y: \text{separate } Y; z: \text{separate } Z \right) \]

**require**

\[ \text{some\_precondition} \]

**local**

\[ my\_y: \text{separate } Y \]
\[ my\_z: \text{separate } Z \]

**do**

\[ x.f \]
\[ my\_y := y \]
\[ x.g \]
\[ my\_z := z \]
\[ s \left( z \right) \]

**end**
Eager Locking Policy

Eager locking policy has two major drawbacks

- Increases danger of **deadlock**

- References **cannot be passed around** without locking the corresponding objects; programmers have no real control over the locks
Selective Locking

Definition (Feature application rule (refined))
Before a feature is applied, its attached formal arguments must be reserved by the supplier, and its precondition must hold.

⇒ Together with the call validity rule the feature application rule ensures atomicity

Definition (Call validity rule)
Call \( \text{exp.f}(a) \) appearing class \( C \) is valid if and only if the following conditions hold:

- \( \text{exp} \) is controlled.
- \( \text{exp}'s \) base class has feature \( f \) exported to \( C \), and the actual arguments \( a \) conform in number and type to the formal arguments of \( f \).
Selective locking

\[ r (x: \text{separate } X; y: \text{separate } Y; z: \text{separate } Z) \]

**require**

\[ \text{some\_precondition} \]

**local**

\[ my_y: \text{separate } Y \]
\[ my_z: \text{separate } Z \]

**do**

\[ x.f \]
\[ my_y := y \]
\[ x.g \]
\[ my_z := z \]
\[ s (z) \]

**end** \[ \Rightarrow \text{increased potential for parallelism} \]
Inheritance and polymorphism

Refined locking mechanism must be compatible with
- inheritance
- polymorphism
- dynamic binding

Definition (Feature redefinition rule (tentative))
The type of a formal argument may be redefined in a descendant from attached to detachable.
The return type of a feature may be redefined from detachable to attached.
Correct feature redefinition

-- in class C
\[ r(x: \text{separate } X) \]
\[ \text{do} \]
\[ \ldots \]
\[ \text{end} \]

-- in descendant class D
\[ r(x: \text{separate } ?X) \]
\[ \text{do} \]
\[ \ldots \]
\[ \text{end} \]

-- in class C
\[ f: \text{?separate } R \]
\[ \text{do} \]
\[ \ldots \]
\[ \text{end} \]

-- in descendant class D
\[ f: \text{separate } R \]
\[ \text{do} \]
\[ \ldots \]
\[ \text{end} \]
Penalising a client

-- in class C
\[ r (x: \text{separate } X) \text{ do } \ldots \text{ end} \]

-- in descendant class D
\[ r (x: \text{separate } X) \text{ do } \ldots \text{ end} \]

-- in class A
\[
\begin{align*}
  c & : C \\
  d & : D \\
  my\_x & : ?\text{separate } X \\
\end{align*}
\]
\[
\begin{align*}
  c & := d \\
  c.r (my\_x) \\
\end{align*}
\]
Penalising a client

-- in class C
\( f: \) separate \( Y \) do ... end

-- in descendant class \( D \)
\( f: \) ? separate \( Y \) do ... end

-- in class \( A \)
\( c: C \)
\( d: D \)
\( r: \) separate \( Y \)

\( c := d \)
\( r := c.f \)
Outstanding problem

Use of Precursor is not always possible

\[
\text{require} \\
\text{some\_precondition} \\
\text{local} \\
\text{my\_y: separate Y} \\
\text{my\_z: separate Z} \\
\text{do} \\
x.f \\
\text{my\_y := y} \\
x.g \\
\text{my\_z := z} \\
s(z) \\
\text{end}
\]
Redefined version of $r$

\[ r \left( x: \text{separate } X; y: \text{?separate } Y; z: \text{?separate } Z \right) \]

\textbf{require else}

\> \textit{new\_precondition}

\textbf{do}

\> \textbf{if} \textit{new\_precondition} \textbf{then}

\> -- do something here

\> \textbf{else} -- here \textit{some\_precondition} holds

\> \textbf{Precursor} \left( x, y, z \right) -- invalid!

\textbf{end}

\textbf{end}
Object test (downcast) necessary

\[ r (x: \text{separate } X; y: \text{separate } Y; z: \text{separate } Z) \]

require else

new_precondition

do

if new_precondition then

-- do something here

elseif \{ aux_y: \text{separate } Y \} y and then \{ aux_z: \text{separate } Z \} z

Precursor (x, aux_y, aux_z) -- valid!

end

end
Pre- and Postcondition

Inherited precondition and postcondition clauses that involve calls on the redefined arguments may become invalid.

-- redefined version of $r$ from slide on page 12

$r (x: \text{separate } X; y: ?\text{separate } Y; z: ?\text{separate } Z)$

require else

new_precondition

do

if new_precondition then
    -- do something here
else -- here some_precondition holds
    -- but what if some_precondition is $y.is\_empty$
    -- what does $y.is\_empty$ mean if $y$ is detachable?
end
end
Definition (Feature redefinition rule)

The type of a **formal argument** may be redefined in a descendant from **attached** to **detachable**, provided that no calls on that argument appear in the inherited post-condition.

The **return type** of a feature may be redefined from **detachable** to **attached**.

Definition (Inherited precondition rule)

Inherited precondition clauses involving calls on **detachable** formal arguments hold vacuously. (trivially true)
Need for lock passing

\[ r(x: \text{separate } X; y: \text{separate } Y) \]

\[ \text{do} \]
\[ \quad x.f \]
\[ \quad x.g(y) \quad \text{-- } x \text{ waits for } y \text{ to become available} \]
\[ \quad y.f \]
\[ \quad \ldots \]
\[ \quad z := x.\text{some}_\text{query} \quad \text{-- Current waits for } \text{some}_\text{query} \text{ to finish} \]
\[\quad \text{-- Deadlock!} \]
\[ \text{end} \]
Need for lock passing

\[ s(x: \text{separate } X) \]

\[
\begin{align*}
do & \\
z & := x.g(\text{Current}) \quad -- \text{Current waits for some_query to finish} \\
& \quad -- \text{Deadlock!}
\end{align*}
\]

end
Lock passing mechanism

- Must not compromise the atomicity guarantees

- Clients must be able to decide to pass or not to pass a lock.

- The mechanism should increase the expressiveness of the language, not restrict is.

- The solution must be simple and well integrated with other language mechanisms
Lock proposal

If a feature call $x.f(a_1, \ldots, a_n)$ occurs in the context of the routine $r$ where some actual argument $a_i$ is controlled, i.e. $a_i$ is attached and locked by $r$, and the corresponding formal argument of $f$ is declared as attached, the client's handler (the processor executing $r$) passes all currently held locks (including the implicit lock itself) to the handler of $x$, and waits until $f$ has terminated. When the execution of $f$ is complete, the client's handler resumes the computation.
Example 1

-- in class C

\[ r \left( x: \text{separate } X; \ y: \text{separate } Y \right) \]

\[ \text{do} \]

\[ x.f \]

\[ x.g \left( y \right) \text{ -- } \textbf{Current} \text{ passes its lock to } x \]

\[ \text{-- and waits until } g \text{ terminates} \]

\[ y.f \]

\[ \ldots \]

\[ z := x.\text{some_query} \text{ -- No deadlock here} \]

\[ \text{end} \]

-- in class X

\[ g \left( y: \text{separate } Y \right) \text{ do } \ldots \text{ end} \]
Example 2

-- in class C

$$s \ (x: \texttt{separate} \ X)$$

\begin{verbatim}
  do
    z := x.g (Current) -- x gets lock on Current
    -- No deadlock here!
  end
\end{verbatim}

-- in class X

$$g \ (c: \texttt{separate} \ C): \ Z$$

\begin{verbatim}
  do
    c.f (...) 
  end
\end{verbatim}
Definition (Feature call semantics (refined))

A feature call $x.f(a)$ results in the following sequence of actions performed by the client’s handler $P_c$:

1. Argument passing: bind the formal arguments of $f$ to the corresponding actual arguments $a$. If any attached formal argument corresponds to a controlled actual argument of a reference type, pass all the currently held locks (including a lock on $P_c$) to the supplier’s handler $P_x$.

2. Feature request: ask $P_x$ to apply $f$ to $x$.
   a. Schedule $f$ for an immediate execution by $P_x$ and wait until it terminates, if any of the following conditions holds:
      • The call is non-separate, i.e. $P_c = P_x$.
      • The call is separate callback, i.e. $P_x$ already held a lock on $P_c$ at the moment of the call.
   b. Otherwise, schedule $f$ to execute after the previous calls on $P_x$.

3. Wait by necessity: if $f$ is a query, wait for its result.

4. Lock revocation: if lock passing occurred in step 1, wait for $f$ to terminate, then revoke the locks from $P_x$. 

Example

-- in class C

my_x: X
my_z: separate Z
my_c: separate C
i: INTEGER

-- in class X:

f (i: INTEGER) do ... end

i := ... 

my_z: separate Z

my_z: separate Z

my_c: separate C

my_c: separate C

i: INTEGER

i := ... 

r (x: separate X; y: separate Y)

do

my_x.f (5)
my_x.g (x)
i := my_x.h (Current)
x.f (10)
x.g (my_z)
x.g (y)
x.m (y)
i := x.h (my_c)
i := x.h (Current)
end
Example

-- in class C
my_x: X
my_z: separate Z
my_c: separate C
i: INTEGER

-- in class X:
f (i: INTEGER) do ... end
g (a: separate ANY) do ... end
h (c: separate C) do c.f (...) end
m (a: ?separate ANY) do ... end

r (x: separate X; y: separate Y)
do
  my_x.f (5) -- non-separate, no wait by necessity, no lock passing
  my_x.g (x) -- non-separate, no wait by necessity, lock passing (vacuous)
i := my_x.h (Current) -- non-separate, wait by necessity, lock passing (vacuous)
x.f (10) -- separate, no wait by necessity, no lock passing
x.g (my_z) -- separate, no wait by necessity, no lock passing
x.g (y) -- separate, no wait by necessity, lock passing
x.m (y) -- separate, no wait by necessity, no lock passing
i := x.h (my_c) -- separate, wait by necessity, no lock passing
i := x.h (Current) -- separate, wait by necessity, lock passing
end
## Lock passing combinations

<table>
<thead>
<tr>
<th></th>
<th>formal attached</th>
<th>formal detachable</th>
</tr>
</thead>
<tbody>
<tr>
<td>actual (reference type) controlled</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>actual (reference type) uncontrolled</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>actual expanded</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

*yes stands for: lock passing takes place*

*no stands for: no lock passing*
Genericity

- Entities of generic types may be separate

  \[ \text{list: } \text{LIST} \ [\text{BOOK}] \]

  \[ \text{list: separate } \text{LIST} \ [\text{BOOK}] \]

- Actual generic parameters may be separate

  \[ \text{list: } \text{LIST} \ [\text{separate BOOK}] \]

  \[ \text{list: separate } \text{LIST} \ [\text{separate BOOK}] \]

- All combinations are meaningful and useful
Genericity: even more fun

- Actual generic parameters may be also of generic type

\[ \text{list: } \text{LIST} [\text{separate SET} [\text{SET} [\text{separate BOOK}]]] \]

- Separateness is relative to object of generic class, e.g. elements of

\[ \text{list: separate LIST [BOOK]} \]

are non-separate with respect to (w.r.t) list but separate w.r.t. Current

- Type combinators apply
Full support

- list: LIST [BOOK]
- list: LIST [separate BOOK]
- list: separate LIST [BOOK] -- BOOK expanded
- list: separate LIST [separate BOOK]

- list: separate <p> LIST [BOOK]
  -- BOOK non-expanded

Equivalent with

- list: separate <p> LIST [separate <p> BOOK]