Concepts of Concurrent Computation

Bertrand Meyer
Sebastian Nanz

Lecture 8: SCOOP advanced concepts
Today’s lecture

In this lecture you will learn about:

• Lock passing, a mechanism implemented in SCOOP for deadlock avoidance
• The changed semantics of contracts in SCOOP, especially that of postconditions
• Inheritance in SCOOP
• Definition and use of agents (function objects) in SCOOP
• The semantics of once functions in SCOOP
EiffelSoftware SCOOP capabilities

- Processor tags: not supported
- Asynchronous postcondition evaluation: not supported
- (Deep) Import operation for expanded types: not supported
- Lock passing: supported
- Separate callbacks: not supported
- Valid feature redeclaration with respect to separateness: supported
- Object tests that incorporate processor locality: not supported
- Agents: not supported
- Once routines: supported
Lock passing
The need for lock passing

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

local
\[ z: \text{separate ANY} \]

\[ \text{do} \]
\[ x.f \]
\[ x.g (y) \]
\[ y.f \]
\[ z := x.\text{some\_query} \]
\[ \text{end} \]

\( y \) is locked by \textbf{Current}  

Waits for \( y \) to become available  

\textbf{Deadlock: wait for} \textbf{some\_query} to finish
Lock passing

... 

x.f

x.g (y)

g (y: separate Y)
do

... 

y.f

... 

y.f

end
Lock passing

- If a call \( x.f(a_1, \ldots, a_n) \) occurs in a routine \( r \) where one or more \( a_i \) are controlled, the client's handler (the processor executing \( r \)) passes all currently held locks to the handler of \( x \), and waits until \( f \) terminates.
- When \( f \) terminates, the client resumes its computation.

\[
\begin{align*}
   &r (x: \text{separate } X; y: \text{separate } Y) \\
   &\quad \text{local} \\
   &\quad \quad z: \text{separate } \text{ANY} \\
   &\quad \text{do} \\
   &\quad \quad x.f \\
   &\quad \quad x.g(y) \\
   &\quad \quad y.f \\
   &\quad \quad z := x.\text{some_query} \\
   &\quad \text{end}
\end{align*}
\]

Pass locks to \( g \) and wait for \( g \) to finish

Synchronous

Synchronous
## Lock passing combinations

<table>
<thead>
<tr>
<th>Formal</th>
<th>Attached</th>
<th>Detachable</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Actual</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reference, controlled</td>
<td>Lock passing</td>
<td>no</td>
</tr>
<tr>
<td>Reference, uncontrolled</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Expanded</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>
Lock passing: example

class $C$ feature
   $x_1$: $X$
   $z_1$: separate $Z$
   $c_1$: separate $C$
   $i$: INTEGER

   $r (x: \text{separate} \ X; y: \text{separate} \ Y)$
      do
         $x_1.f (5)$
         $x_1.g (x)$
         $i := x_1.h (\text{Current})$
         $x.f (10)$
         $x.g (z_1)$
         $x.g (y)$
         $x.m (y)$
         $i := x.h (c_1)$
         $i := x.h (\text{Current})$
      end
   end

class $X$ feature
   $f (i: \text{INTEGER})$ do ... end
   $g (a: \text{separate} \ ANY)$ do ... end
   $h (c: \text{separate} \ C): \text{INTEGER}$ do $c.p (...) \end do$
   $m (a: \text{detachable separate} \ ANY)$ do ... end

Non-separate, no wait by necessity, no lock passing
Non-separate, no wait by necessity, lock passing (vacuous)
Non-separate, wait by necessity, lock passing (vacuous)
Separate, no wait by necessity, no lock passing
Separate, no wait by necessity, no lock passing
Separate, no wait by necessity, no lock passing
Separate, no wait by necessity, lock passing
Separate, no wait by necessity, no lock passing
Separate, wait by necessity, no lock passing
Contracts
Preconditions

• In sequential context: precondition is correctness condition
• In concurrent context: feature call and feature application do not usually coincide
  • A supplier cannot assume that a property satisfied at the call time still holds at the execution time.
Preconditions

store (b: separate BUFFER [INTEGER]; i: INTEGER)
  -- Store i in buffer.
  require
    not b.is_full
    i > 0
do
  b.put (i)
end

my_buffer: separate BUFFER [INTEGER]
ns_buffer: BUFFER [INTEGER]
...
store (my_buffer, 24)
store (ns_buffer, 24)
my_buffer := ns_buffer
store (my_buffer, 79)
Preconditions

- A precondition expresses the necessary requirements for a correct feature application.
- Precondition viewed as synchronization mechanism:
  - A called feature cannot be executed unless the preconditions hold
  - A violated precondition delays the feature’s execution
- The guarantee given to the supplier is exactly the same as with the traditional semantics.
Postconditions

- A postcondition describes the result of a feature's application.
- Postconditions are evaluated asynchronously; wait by necessity does not apply.
- After returning from the call the client can only assume the controlled postcondition clauses.
Postconditions

\[\text{spawn}_\text{two}(l_1, l_2: \text{separate LOCATION})\]
\[\text{do}\]
\[l_1.\text{do\_job}\]
\[l_2.\text{do\_job}\]
\[\text{ensure}\]
\[\text{post}_1: l_1.\text{is\_ready}\]
\[\text{post}_2: l_2.\text{is\_ready}\]
\[\text{end}\]

tokyo, zurich: \text{separate LOCATION}

\[r(l: \text{separate LOCATION})\]
\[\text{do}\]
\[\text{spawn}_\text{two}(l, \text{tokyo})\]
\[\text{do\_local\_stuff}\]
\[\text{get\_result}(l)\]
\[\text{do\_local\_stuff}\]
\[\text{get\_result}(\text{tokyo})\]
\[\text{end}\]

\[\ldots\]
\[r(\text{zurich})\]
Inheritance
Inheritance

• Can we use inheritance as in the sequential world?
• Is multiple inheritance allowed?
• Does SCOOP suffer from inheritance anomalies?
Example: Dining Philosophers

class PHILOSOPHER inherit GENERAL_PHILOSOPHER
    PROCESS
        rename
            setup as getup
        undefine
            getup
        end
feature
    step
        -- Perform a philosopher's tasks.
        do
            think ; eat (left, right)
        end

    eat (l, r: separate FORK)
        -- Eat, having grabbed l and r.
        do ... end
end
Dining Philosophers

defered class PROCESS feature
    over: BOOLEAN
        -- Should execution terminate now?
defered end

setup
    -- Prepare to execute process operations.
defered end

step
    -- Execute basic process operations.
defered end

wrapup
    -- Execute termination operations (default: nothing).
do end

live
    -- Perform process lifecycle.
do
    from setup until over loop
        step
    end
wrapup
end
Dining Philosophers

class GENERAL_PHILOSOPHER create
    make
feature -- Initialization
    make (l, r: separate FORK)
        -- Define l as left and r
        -- as right forks.
    do
        left := l
        right := r
    end
end

class FORK
end

feature {NONE} -- Implementation
    left: separate FORK
    right: separate FORK

    getup
        -- Take initialization actions.
        do end

    think
        -- Philosopher's act.
        do end
end
Inheritance

- Full support for inheritance (including multiple inheritance)
- Most inheritance anomalies eliminated thanks to the proper use of OO mechanisms
Inheritance and Contracts

• Preconditions may be kept or weakened.
  • Less waiting
• Postconditions may me kept or strengthened.
  • More guarantees to the client
• Invariants may be kept or strengthened
  • More consistency conditions
Inheritance: Result type redeclaration (functions)

class C feature
  r (x: X)
    do ... end
  s (y: separate Y)
    do ... end
end

class A feature
  x: X
  y: separate Y
end

class B inherit A redefine x, y end

feature
  x: separate X
  y: Y
end

--Would lead to a traitor:
c: C a: A
create {B} a
c.r (a.x)

-- This one is OK:
c: C a: A
create {B} a
c.s (a.y)

• Result types may be redefined covariantly for **functions**. For **attributes** the result type may not be redefined.
Inheritance: formal argument redeclaration

- Formal argument types may be redefined contravariantly w.r.t. processor tags.
Inheritance: formal argument redeclaration

```
class A feature
  r (x: detachable separate X)
    do ...
  end

  s (x: separate X)
    do ...
  end
end

class B inherit
  A redefine r, s end

feature
  r (x: separate X)
    do ...
  end

  s (x: detachable separate X)
    do ...
  end
end
```

- Formal argument types may be redefined contravariantly w.r.t detachable tags. The client waits less.

Additional locking for client: not acceptable

Less locking for client: acceptable
Agents
What is an agent?

- An agent represents an operation ready to be called.
  
  \[
  x: X \\
  \text{op1: ROUTINE} \ [X, \ \text{TUPLE}] \\
  \text{op1} := \text{agent} \ x.f \\
  \text{op1}.\text{call} \ ([])
  \]

- Agents can be created by one object, passed to another one, and called by the latter
What is an agent?

• Arguments can be closed (fixed) or open.
  
  \[
  \text{op1} := \text{agent} \ \text{io.put_string} \ ("Hello World!")
  \]
  \[
  \text{op1} \ . \text{call} \ ([])
  \]
  
  \[
  \text{op1} := \text{agent} \ \text{io.put_string} \ (?)
  \]
  \[
  \text{op1} \ . \text{call} \ (["Hello World!"])
  \]

• They are based on generic classes:

ROUTINE [BASE_TYPE, OPEN_ARGS -> TUPLE]
PROCEDURE [BASE_TYPE, OPEN_ARGS -> TUPLE]
FUNCTION [BASE_TYPE, OPEN_ARGS -> TUPLE, RESULT_TYPE]
Use of agents

Object-oriented wrappers for operations
- Strongly-typed function pointers (C++)
- Similar to .NET delegates

Used in event-driven programming
- Subscribe an action to an event type
- The action is executed when event occurs

Loose coupling of software components

Replace several patterns
- Observer
- Visitor
- Model - View - Controller

...
Problematic agents

- Which processor should handle an agent? Is it the target processor or the client processor?
- Let's assume it is the client processor.

\[ P_1 \quad \text{target} \quad P_2 \]

\[
\begin{align*}
\text{a1: PROCEDURE } & \text{[separate ANY, TUPLE]} \\
x: \text{separate } X \\
\ldots \\
a1 := \text{agent } x.f \\
a1.\text{call } ([]) \\
\end{align*}
\]

Like \( x.f \) without locking \( x \)

x

\[
\begin{align*}
\text{separate reference} \\
\text{non-separate reference} \\
\end{align*}
\]
Let’s make the agent separate!

- The agent needs to be on the target processor.

\[ a1 := \text{agent } x.f \]

This agent will be handled by \( x \)'s processor

Invalid
Let’s make the agent separate!

• No special type rules for separate agents
• Semantic rule: an agent is created on its target’s processor
• Agents pass processors’ boundaries just as other objects do

\[
\begin{align*}
\text{a1: separate } & \text{PROCEDURE } [X, \text{TUPLE}] \\
x: & \text{separate } X \\
a1 := & \text{agent } x.f
\end{align*}
\]

call (a1)

call (an_agent: separate PROCEDURE [ANY, TUPLE])
\[
\begin{align*}
do & \\
\text{an_agent.call ([]} & \text{Valid separate call}
\end{align*}
\]
end
First benefit: convenience

- Without agents, enclosing routines are necessary for every separate call.

```c
x1: separate X
r (x: separate X) do
s (x: separate X) do
  x.f
  x.g (5, "Hello")
end
end
```

- With agents, we can write a universal enclosing routine.

```c
call (agent x1.f); call (agent x1.g (5, "Hello"))
call (an_agent: separate PROCEDURE [ANY, TUPLE])
  -- Universal enclosing routine.
  do
    an_agent.call ([]) end
```
Second benefit: full asynchrony

- Without agents, full asynchrony cannot be achieved
  \[
  \begin{align*}
  &x_1, y_1: \text{separate} \ X \\
  &r \ (x: \text{separate} \ X) \\
  &r \ (x_1) \\
  &\text{do}\_\text{local}\_\text{stuff} \\
  &\text{do} \\
  &\quad x.f \\
  &\text{end}
  \end{align*}
  \]

- With agents it works
  \[
  \begin{align*}
  &\text{async} \ (\text{agent} \ x_1.f) \\
  &\text{do}\_\text{local}\_\text{stuff} \\
  &\text{async} \ (a: \text{detachable} \ \text{separate} \ \text{PROCEDURE} \ [\text{ANY}, \ \text{TUPLE}]) \\
  &\quad -- \text{Call } a \text{ asynchronously.} \\
  &\text{do} \\
  &\quad \ldots \\
  &\text{end}
  \end{align*}
  \]
Full asynchrony

The feature *asynch* can be implemented as follows:

```plaintext
asynch (a: detachable separate PROCEDURE [ANY, TUPLE])
    -- Call a asynchronously.
    -- Note that a is not locked.
local
    executor: separate EXECUTOR
do
    create executor.make (a)
    launch (executor)
end
```

An asynchronous call on a non-separate targets (including *Current*) will be executed when the current processor becomes idle.
Third benefit: waiting faster

\[ \begin{align*}
x_1, y_1: & \text{ separate } X \\
o_r \_e_s (x, y: \text{ separate } X): & \text{ BOOLEAN} \\
& \text{ do} \\
& \text{ if } o_r \_e_s (x_1, y_1) \text{ then} \\
& \quad \ldots \\
& \text{ end} \\
& \text{ Result } := x.b \text{ or else } y.b \\
& \text{ end}
\end{align*} \]

- What if \( x_1 \) or \( y_1 \) is busy?
- What if \( x_1.b \) is false but \( y_1.b \) is true?
- What if evaluation of \( x_1.b \) takes ages whereas \( y_1.b \) evaluates very fast?
Waiting faster

if parallel_or (agent x1.b, agent y1.b) then

... end

parallel_or (a1, a2: detachable separate FUNCTION [ANY, TUPLE, BOOLEAN]): BOOLEAN

-- Result of a1 or else a2 computed in parallel.

local

ans_col: separate ANSWER_COLLECTOR [BOOLEAN]
do

create ans_col.make (a1, a2)

Result := answer (ans_col)
end

answer (ac: separate ANSWER_COLLECTOR [BOOLEAN]): BOOLEAN

-- Result returned by ac.

require

answer_ready: ac.is_ready
do

Result := ac.answer
end
Agents wrap-up

- **Agents and concurrency**
  - Tricky at first; easy in the end
  - Agents built on separate calls are separate
  - Agents treated just like any other object

- **Advantages brought by agents**
  - Convenience: “universal” enclosing routine for single calls
  - Full asynchrony: non-blocking calls
  - Truly parallel wait
Once functions
Once Functions

- Similar to constants
  - Always return the same value
- Lazy evaluation
  - Body executed on first access
- Once per thread or once per object semantic
- Examples of use
  - Heavy computations
    - Stock market statistics
  - Common contact point for objects of one type
  - Feature io in class ANY
Once functions in a concurrent context

- Is once-per-system semantics always correct?
  
  ```
  barrier: separate BARRIER
  once
  create Result.make (3)
  end
  ```

  ```
  local_printer: PRINTER
  once
  printer_pool.item (Current.location)
  end
  ```

- Separate functions are once-per-system.
- Non-separate functions are once-per-processor.