Concurrent Object-Oriented Programming

Bertrand Meyer, Volkan Arslan
Lecture 12: SCOOP: Advanced techniques - Inheritance and Contracts
Outline

SCOOP concurrency model

- Inheritance
- Contracts
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 is
    -- Perform a philosopher's tasks.
    do
      think
      eat (left, right)
    end

  eat (l, r: separate FORK) is
    -- Eat, having grabbed l and r.
    do
      ...
    end
end -- class PHILOSOPHER
Class `PROCESS`

indexed

description: "The most general notion of process"

**deferred class** `PROCESS`

**feature** -- Status report

`over: BOOLEAN is`

`-- Should execution terminate now?`

`deferred end`

**feature** -- Basic operations

`setup is`

`-- Prepare to execute process operations (default: nothing).`

`do end`

`step is`

`-- Execute basic process operations.`

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

feature -- Process behavior

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

end -- class PROCESS
Class GENERAL PHILOSOPHER

class

GENERAL PHILOSOPHER

create

make

feature -- Initialization

make (l, r: separate FORK) is
    -- Define l as left and r as right forks.
    do
        left := l
        right := r
    end

feature {NONE} -- Implementation

left: separate FORK

right: separate FORK
Class GENERAL_PHILOSOPHER (cont.)

getup is
    -- Take any necessary initialization action.
    do
    end

think is
    -- Any appropriate action or lack thereof
    do
    end

end -- class GENERAL_PHILOSOPHER

class FORK
end
Inheritance

- Full support for inheritance (including multiple inheritance)

- Usual rules apply
  - Weakening pre-conditions
  - Strengthening post-conditions

- Most inheritance anomalies eliminated thanks to the proper use of OO mechanisms

- What’s about Active objects?
Contracts in SCOOP

- Preconditions
- Postconditions
- Invariants
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
store (buffer: separate BUFFER [INTEGER]; i: INTEGER) is
  -- Store i in buffer.
  require
    not buffer.is_full
    i > 0
  do
    buffer.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

- Sometimes: a wait condition should be turned into correctness condition
- Sometimes: a correctness condition should be turned into a wait condition

Precondition viewed as **synchronization** mechanism:
- a called feature cannot be executed **unless the preconditions hold**
- a violated precondition **delays** the feature execution
Preconditions

Definition (Semantics of precondition)

A precondition expresses the necessary requirements for a correct feature application. The execution of the feature’s body is delayed until the precondition is satisfied.

⇒ Guarantee given to the supplier is exactly the same as with the traditional semantics.
Postconditions

Definition (Semantics of postconditions)

- A postcondition describes the result of a feature’s application.
- Postconditions are evaluated *asynchronously*; wait by necessity *does not* apply.
- Postcondition clauses that do not involve calls on objects handled by the same processors are evaluated *independently*. 
Postconditions

\[
\text{spawn\_two\_activities (location\_1, location\_2: separate LOCATION) is}
\]
\[
\quad \text{do}
\quad \quad \text{location\_1.do\_job}
\quad \quad \text{location\_2.do\_job}
\]
\[
\quad \text{ensure}
\quad \quad \text{post\_1: location\_1.is\_ready}
\quad \quad \text{post\_2: location\_2.is\_ready}
\]
\[
\quad \text{end}
\]

\[
tokyo: \text{separate LOCATION}
\]
\[
\text{my\_york: separate LOCATION}
\]

\[
r (york: separate LOCATION) is
\]
\[
\quad \text{do}
\quad \quad \text{spawn\_two\_activities (york, tokyo)}
\quad \quad \text{do\_local\_stuff}
\quad \quad \text{get\_result (york)}
\quad \quad \text{do\_local\_stuff}
\quad \quad \text{get\_result (tokyo)}
\]
\[
\quad \text{end}
\]
\[
\quad \ldots
\]
\[
r (my\_york)
\]
Invariants

- Standard Eiffel semantics applies to class invariants in SCOOP because all the calls appearing in invariants must be **non-separate**

- **Asynchronous** semantics is applied to loop variants, loop invariants, and check instruction.
remove_n (list: separate LIST[G]; n: INTEGER) is
   -- Remove n elements from list.
   require
     list.count >= n
   local
     initial, removed: INTEGER
   do
     from
       initial := list.count
       removed := 0
     until
       removed = n
     invariant
       list.count + removed = initial
     variant
       list.count - initial + n       -- the same as n - removed
     loop
       list.remove
       removed := removed + 1
     end
   ensure
     list.count = old list.count + 1
   end