Concurrent Object-Oriented Programming

Bertrand Meyer

Lecture 2: Overview of SCOOP

Updated: 03 April 2006

Chair of Software Engineering
Basic goal

Can we bring concurrent programming to the same level of abstraction and convenience as sequential programming?
SCOOP in a nutshell

- No intra-object-concurrency
- One keyword: separate, indicates thread of control is “elsewhere”
- Reserve one or more objects through argument passing
- Preconditions become wait conditions
- Exception-based mechanism to break lock
**my_queue**: BUFFER [T]

...  

if not **my_queue**.is_full then  

**store**(my_queue, t)  

end

**store**(b: BUFFER [G]; v: G)  

-- Store v into b.

**require**  

not b.is_full  

**do**  

...  

**ensure**  

not b.is_empty  

end
Data races and other delights of life


```java
class ResourceStoreManager {
    boolean closed = false;
    Map entries = new HashMap();

    synchronized void checkClosed() {
        if (closed)
            throw new RuntimeException();
    }

    ResourceStore loadResourceStore(...) {
        checkClosed();
        StoreEntry se = lookupEntry(...);
        return se.getStore();
    }

    synchronized Entry lookupEntry(...) {
        Entry e = (Entry) entries.get(...);
        if (e == null) {
            e = new Entry();
            entries.put(..., e);
        }
        return e;
    }

    synchronized void shutdown() {
        while (...) {
            // remove all entries
            closed = true;
        }
    }
}
```
Dining philosophers

class PHILOSOPHER inherit PROCESS
  rename setup as getup
  redefine step end

feature {BUTLER}
  step
    do
      think; eat(left, right)
    end

  eat(l, r: separate FORK)
    -- Eat, having grabbed l and r.
    do ... end
end
The big wide world of concurrency

Multithreading
Internet-based applications
Distribution
Pervasive computing
Web services
(Coroutines...)
## Previous advances in programming

<table>
<thead>
<tr>
<th>Feature</th>
<th>&quot;Structured programming&quot;</th>
<th>&quot;Object technology&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use higher-level abstractions</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Helps avoid bugs</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Transfers tasks to implementation</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Lets you do stuff you couldn't before</td>
<td>NO</td>
<td>✓</td>
</tr>
<tr>
<td>Removes restrictions</td>
<td>NO</td>
<td>✓</td>
</tr>
<tr>
<td>Adds restrictions</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Has well-understood math basis</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Doesn't require understanding that basis</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Permits less operational reasoning</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
Then and now

**Sequential programming:**

- Used to be messy
- Still hard but:
  - Structured programming
  - Data abstraction & object technology
  - Design by Contract
  - Genericity, multiple inheritance
  - Architectural techniques

Switch from operational reasoning to logical deduction (e.g. invariants)

**Concurrent programming:**

- Used to be messy
- Still messy

Example: threading models in most popular approaches

Development level: sixties/seventies

Only understandable through operational reasoning
This mechanism

SCOOP: Simple Concurrent Object-Oriented Programming

First iteration 1990 -- CACM, 1993

Object-Oriented Software Construction, 2nd edition, 1997

Prototype implementations, 1995-now

Now being done for good at ETH thanks to this project

On top of Eiffel Software's compiler (native Windows, .NET)
Can object technology help?

“Objects are naturally concurrent” (Milner)

Many attempts, often based on (self-contradictory) notion of “Active objects”

Often lead to “Inheritance anomaly”

None widely accepted

In practice: low-level mechanisms on top of O-O language
Object-oriented computation

To perform a computation is

- To apply certain **actions**
- To certain **objects**
- Using certain **processors**
What makes an application concurrent?

**Processor:**
Thread of control supporting sequential execution of instructions on one or more objects

Can be implemented as:
- Computer CPU
- Process
- Thread
- AppDomain (.NET) ...

Will be mapped to computational resources
Handling rule

All calls on an object are executed by the processor’s handler
Reasoning about objects

\[
\{\text{Pre}_r \text{ and INV}\} \quad \text{body}_r \quad \{\text{Post}_r \text{ and INV}\}
\]

\[
\{\text{Pre}_r'\} \quad x.r \ (a) \quad \{\text{Post}_r'\}
\]
Reasoning about objects

Only $n$ proofs if $n$ exported routines!

$\{\text{Pre}_r \text{ and INV}\} \ \text{body}_r \ \{\text{Post}_r \text{ and INV}\}$

$\{\text{Pre}_r'\} \ x.r \ (a) \ \{\text{Post}_r'\}$
In a concurrent context

Only $n$ proofs if $n$ exported routines?

\[
\{ \text{Pre}_r \ \text{and INV} \} \ \text{body}_r \ \{ \text{Post}_r \ \text{and INV} \}
\]

\[
\{ \text{Pre}_r' \} \ x.r \ (a) \ \{ \text{Post}_r' \}
\]

Client 1, $r_1$  
Client 2, $r_2$  
Client 3, $r_3$
Mutual exclusion rule

At most one feature may execute on any one object at any one time
Feature call: sequential

\[ x.r(a) \]

\( x \in CX \)

**Client**

previous_instruction

\( x.r(a) \)

next_instruction

**Supplier (CX)**

\( r(a : A) \)

require

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

ensure

not \( a \cdot \text{is_empty} \)

end

**Processor**
Feature call: asynchronous

\[ x \cdot r(a) \]

\( x \): separate \( CX \)

**Client**
- previous_instruction
- \( x \cdot r(a) \)
- next_instruction

**Supplier (\( CX \))**
- \( r(a : A) \)
- require \( a / \neq \text{Void} \)
- ensure not \( a \cdot is\_empty \)
- end

Client’s processor \( \rightarrow \) Supplier’s processor

\( \text{Void} \)
Separateness rule

Calls on non-separate objects are blocking

Call on separate objects are non-blocking
Feature call: asynchronous

\[ x.r(a) \]

\( x: \) separate \( CX \)

Client

\begin{align*}
\text{previous\_instruction} \\
\text{x.r(a)} \\
\text{next\_instruction}
\end{align*}

Client processor

Supplier (\( CX \))

\begin{align*}
r(a: A) \\
\text{require} \\
\text{a} \neq \text{Void} \\
\text{ensure} \\
\text{not} \ a \cdot \text{is\_empty} \\
\text{end}
\end{align*}

Supplier processor

\( \text{end} \)
The fundamental difference

To wait or not to wait:
If same processor, synchronous
If different processor, asynchronous

Difference must be captured by syntax:

- \textit{x: CX}
- \textit{x: separate CX} \quad -- \textit{potentially} different processor
Consistency

Client:

```ruby
class C feature
  my_a: SOME_TYPE
  b: separate B
  b.p (my_a)
end
```

Supplier:

```ruby
class B feature
  p (a: SOME_TYPE)
  do ... end
end
```
Consistency

Client:

class C feature

my_a: SOME_TYPE

b: separate B

b.p (my_a)

end

Supplier:

class B feature

p (a: separate SOME_TYPE)

do ... end

end
Separateness consistency rule

For any reference actual argument in a separate call, the corresponding formal argument must be declared as separate

Separate call: $a.f(...)$ where $a$ is separate
If no access control

\[ my\_stack: \text{separate STACK}[T] \]

\[ \ldots \]

\[ my\_stack.push(a) \]

\[ y := my\_stack.top \]
Access control policy

Require target of separate call to be formal argument of enclosing routine:

```plaintext
put (stack: separate STACK[T]; value: T)
   -- Push value on top of stack.
   do
      stack.push (value)
   end
```
Access control policy

Target of a separate call must be formal argument of enclosing routine:

\[
\text{store (buffer: separate BUFFER [T]; value: T)}
\]

\[
\quad \text{-- Store value into buffer.}
\]

\[
\text{do}
\]

\[
\text{buffer.put (value)}
\]

\[
\text{end}
\]

To use separate object:

\[
\text{my_buffer: separate BUFFER [INTEGER]}
\]

\[
\text{create my_buffer}
\]

\[
\text{store (my_buffer, 10)}
\]
Separate argument rule

The target of a separate call must be an argument of the enclosing routine

Separate call: $x.f(...)$ where $x$ is separate
A routine call with separate arguments will execute when all corresponding objects are available

and hold them exclusively for the duration of the routine
store (buffer: BUFFER [INTEGER]; v: INTEGER)
-- Store v into buffer.
require
not buffer.is_full
v > 0
do
  buffer.put (v)
ensure
not buffer.is_empty
end
...
store (my_buffer, 10)
my_queue: BUFFER[T]

if not my_queue.is_full then
  store(my_queue, t)
end
From preconditions to wait-conditions

store (buffer: separate BUFFER [INTEGER]; v: INTEGER)
  -- Store v into buffer.
  require
    not buffer.is_full
    v > 0
  do
    buffer.put (v)
  ensure
    not buffer.is_empty
  end
...
store (my_buffer, 10)

On separate target, precondition becomes wait condition
Separate precondition rule

A separate precondition causes the client to wait

Separate precondition: \texttt{a.condition (...)}
where \texttt{a} is separate
Full synchronization rule

A call with separate arguments waits until:
- The corresponding objects are all available
- Separate preconditions hold

\[ x.f(a) \]

where \( a \) is separate
Resynchronization

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

The client will wait only when it needs to:

\[
\begin{align*}
  & x.f \\
  & x.g \ (a) \\
  & y.f \\
  & \ldots \\
  & \text{value} := x \_ \text{some\_query}
\end{align*}
\]
Resynchronization rule

Clients wait for resynchronization on queries
### Duels

#### Library features

<table>
<thead>
<tr>
<th>Challenger →</th>
<th>normal_service</th>
<th>immediate_service</th>
</tr>
</thead>
<tbody>
<tr>
<td>↓ Holder</td>
<td></td>
<td></td>
</tr>
<tr>
<td>retain</td>
<td>Challenger waits</td>
<td>Exception in challenger</td>
</tr>
<tr>
<td>yield</td>
<td>Challenger waits</td>
<td>Exception in holder; serve challenger</td>
</tr>
</tbody>
</table>

**Challenger** waits when there is an exception in the holder; the **holder** waits when there is an exception in the challenger.
Other aspects

- What does a separate postcondition mean?
  \[
  r(a: \text{separate } T)
  ...
  \begin{align*}
  &\text{ensure} \\
  &\quad \textbf{not } a.is\_empty
  \end{align*}
  \]

- What if a separate call, e.g. in
  \[
  r(a: \text{separate } T)
  \begin{align*}
  &\text{do} \\
  &\quad a.f \\
  &\quad a.g \\
  &\quad a.h \\
  &\text{end}
  \end{align*}
  \]
  cause an exception?
Tentative proof rule

\[
\begin{align*}
\{ INV \land Pre_r \} \quad & body_r \quad \{ INV \land \forall_i \Diamond Post^i_r \} \\
\{ \Diamond (Acq(\bar{a}) \land Pre_r[\bar{a} / \bar{x}]) \} \quad & r(\bar{a}) \quad \{ \forall_i (\Diamond Rel(a^i) \land \neg Rel(a^i) \cup Post^i_r[\bar{a} / \bar{x}]) \}
\end{align*}
\]
Implementation: two-level architecture

Adaptable to many environments
Currently implemented for native Windows (using POSIX threads) and .NET
SCOOPLI: Library for SCOOP

Library-based solution

Implemented in Eiffel

Preprocessor and type checker
Elevator example architecture

For maximal concurrency, all objects are separate

Inheritance
Class BUTTON

separate class

    BUTTON

feature

    target: INTEGER

end
Class `CABIN_BUTTON`

```plaintext
separate class `CABIN_BUTTON` inherit `BUTTON`  

feature  
  `cabin: ELEVATOR`

  `request`
    -- Send to associated elevator a request to stop on level `target`.  
    do  
      `actual_request (cabin)`  
    end

  `actual_request (e: ELEVATOR)`  
    -- Get hold of `e` and send a request to stop on level `target`.  
    do  
      `e.accept (target)`  
    end

end
```

Chair of Software Engineering  
Concurrent Object-Oriented Programming
Class **ELEVATOR**

separate class **ELEVATOR** feature \{**BUTTON, DISPATCHER**\}

    accept (floor: INTEGER)
        -- Record and process a request to go to floor.
        do
            record (floor)
            if not moving then process_request end
        end

feature \{**MOTOR**\}

    record_stop (floor: INTEGER)
        -- Record information that elevator has stopped on floor.
        do
            moving := False ; position := floor ; process_request
        end
Class ELEVATOR

feature {NONE} -- Implementation
    process_request
        -- Handle next pending request, if any.
        local floor: INTEGER do
            if not pending.is_empty then
                floor := pending.item; actual_process (puller, floor)
                pending.remove
            end
        end

    actual_process (m: MOTOR, floor: INTEGER)
        -- Handle next pending request, if any.
        do
            moving := true; m.move (floor)
        end

feature {NONE} -- Implementation
    puller: MOTOR; pending: QUEUE [INTEGER]
end
Class *MOTOR*

```plaintext
separate class MOTOR feature {ELEVATOR}
  move (floor: INTEGER)
    -- Go to floor, once there, report.
    do
      gui_main_window.move_elevator (cabin_number, floor)
      signal_stopped (cabin)
    end
  signal_stopped (e: ELEVATOR)
    -- Report that elevator e stopped on level position.
    do  e.record_stop (position) end
feature {NONE}
  cabin: ELEVATOR ; position: INTEGER  -- Current floor level.
  gui_main_window: GUI_MAIN_WINDOW
end
```
Why SCOOP?

**SCOOP model**

- Simple yet powerful
- Easier and safer than common concurrent techniques, e.g. Java Threads
- Full concurrency support
- Full use O-O and Design by Contract
- Supports various platforms and concurrency architectures
- One new keyword: `separate`

**Tools**

- SCOOPILI library
- Pre-processor and type checker
- Full integration with the compiler coming soon
Why SCOOP?

Extend object technology with general and powerful concurrency support

Provide the industry with simple techniques for parallel, distributed, internet, real-time programming

Make programmers sleep better!
Status

- All of SCOOP except duels implemented
- Preprocessor and library available for download
- Numerous examples available for download

[se.ethz.ch/research/scoop.html](http://se.ethz.ch/research/scoop.html)

We are very grateful to the Hasler Foundation for their support.
Lessons

- Concurrency does come naturally to the O-O world
- Must revise usual modes of reasoning about programs
- Design by Contract the key
- A simple extension is possible
- The mechanism can be quite general
- SCOOP is here today, try it!
- We can bring concurrent programming to the same level of safety and elegance as traditional sequential programming
- We don't really have a choice!
Current developments & open problems

Semantic specification
Type system for eliminating atomicity violations
Distribution and web services
Support for transactions
Deadlock prevention and detection
Wait on first of several events
Extensions for real-time
Integration with compiler

se.ethz.ch/research/scoop.html