Concurrent Object-Oriented Programming

Bertrand Meyer

Lecture 2: Overview of SCOOP
The basic goal

Can we bring concurrent programming to the same level of abstraction and convenience as sequential programming?
put
put (b : BUFFER[T]; v : T)  
-- Store v into b.

require
not b.is_full

do
...

ensure
not b.is_empty
end
The issue

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
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
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;
}
```
The big wide world of concurrency

Multithreading
Internet-based applications
Distribution
Pervasive computing
Web services
Multi-core
(Coroutines...)
<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

On top of Eiffel Software’s compiler and EiffelThreads library (native Windows, .NET, Unix, Linux...)

Chair of Software Engineering
ETH
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 object’s handler
Reasoning about objects

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

{Pre$_{r'}$} x.$r$ (a) {Post$_{r'}$}
Reasoning about objects

Only \( n \) proofs if \( n \) exported routines!

\[
\begin{align*}
\{\text{INV and Pre}_r\} & \quad \text{body}_r \quad \{\text{INV and Post}_r\} \\
\{\text{Pre}_r\} & \quad x.r(a) \quad \{\text{Post}_r\}
\end{align*}
\]
In a concurrent context

Only \( n \) proofs if \( n \) exported routines?

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

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

No overlapping!
Mutual exclusion rule

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

\[ x : X \]

\[ x.r(a) \]

**Client**

*previous_instruction*

*next_instruction*

**Supplier (X)**

\[ r(a : A) \]

*require*

\[ a /= Void \]

*ensure*

\[ not a.is_empty \]

*end*

**Processor**
Feature call: asynchronous

\[ x.r(a) \]

\( x : \text{separate } X \)

Client

- previous_instruction
- \( x.r(a) \)
- next_instruction

Client's processor

Supplier (X)

- \( r(a:A) \)
  - require \( a \neq \text{Void} \)
  - ensure not \( a \cdot \text{is_empty} \)
- end

Supplier's processor
Separateness rule

Calls on non-separate objects are blocking

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

\( x \): separate \( X \)

\( x.r(a) \)

**Client**

- previous_instruction
- \( x.r(a) \)
- next_instruction

**Supplier (\( X \))**

- \( r(a : A) \)
- require \( a \neq \text{Void} \)
- ensure not \( a \cdot \text{is_empty} \)
- end

**Client processor**

**Supplier processor**
The fundamental difference

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

Difference must be captured by syntax:

- $x: X$
- $x: \text{separate } X$  -- potentially different processor
class C feature

  nonsep: SOME_TYPE

  sep: separate SOME_TYPE

  nonsep := sep

  nonsep • p (a)

end

Traitor!
No-traitors rule

If the source of an attachment is separate, so must the target be.

Attachment: assignment or argument passing)
Consistency

Client:

```ruby
class C feature
  a: SOME_TYPE
  sep: separate B
  sep.p(a)
end
```

Supplier:

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

Client:

```ruby
class C feature
  a: SOME_TYPE
  sep: separate B
  sep.p (a)
end
```

Supplier:

```ruby
class B feature
  p (a: separate SOME_TYPE)
  is do ... a.g ... 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: separate STACK[ T]  

...  

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

```
store (buffer: separate BUFFER [T]; value: T)
  -- Store value into buffer.
  do
    buffer.put (value)
  end
```

To use a separate object:

```
my_buffer: separate BUFFER [INTEGER]
create my_buffer
store (my_buffer, 10)
```
Separate argument rule

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

Separate call: $x.f(...) \text{ where } x \text{ is separate}$
A routine call with separate arguments will execute when all corresponding processors are available and hold them exclusively for the duration of the routine.
Contracts in Eiffel

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)
store $b : BUFFER[G] ; v : G$)

-- Store $v$ into $b$.
require

not $b$.is_full

do

...  

ensure

not $b$.is_empty
end

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)
A precondition causes the client to wait
Full synchronization rule

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

\[ x.f(a) \]
where \( a \) is separate
Resynchronization

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

The client will wait only when it needs to:

\[ x.f \]
\[ x.g(a) \]
\[ y.f \]
\[ ... \]

\[ value := x.some_query \]

Lazy wait (Denis Caromel, wait by necessity)
Resynchronization rule

Clients wait for resynchronization on queries
class CLIENT
feature
  york, tokyo: separate LOCATION
...
spawn_two_activities (l1, l2: separate LOCATION)
  do
    l1.do_job
    l2.do_job
  ensure
    l1.is_ready
    l2.is_ready
end
...
spawn_two_activities (york, tokyo)
do_local_stuff
get_result (york)
...

Semantics of postconditions

Asynchronous evaluation; processor(s) available when and if postcondition holds.

Each clause evaluated individually.

Wait for york only.
Generalised semantics of postconditions

- Each locked processor released only when related postcondition clauses hold.

- Each postcondition clause is evaluated individually.

  ```
  ensure
  location_1.is_ready
  location_2.is_ready
  ```

  is different from

  ```
  ensure
  location_1.is_ready and location_2.is_ready
  ```

- This semantics boils down to correctness semantics for non-separate postconditions.
# Duels

## Library features

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

<table>
<thead>
<tr>
<th>retain</th>
<th>Challenger waits</th>
<th>Exception in challenger</th>
</tr>
</thead>
<tbody>
<tr>
<td>yield</td>
<td>Challenger waits</td>
<td>Exception in holder; serve challenger</td>
</tr>
</tbody>
</table>
### 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>Challenger waits</td>
<td>Exception in challenger</td>
</tr>
<tr>
<td>retain</td>
<td>Challenger waits</td>
<td>Exception in holder; serve challenger</td>
</tr>
<tr>
<td>yield</td>
<td>Challenger waits</td>
<td>Exception in challenger</td>
</tr>
</tbody>
</table>
Other aspects

- What if a separate call, e.g. in

\[
\begin{align*}
r(a & : \text{separate T}) \\
do & \quad a.f \\
d & \quad a.g \\
& \quad a.h \\
end
\end{align*}
\]

cause an exception?
Refined proof rule (partial correctness)

\[
\{ INV \land Pre_r (x) \} \quad body_r \quad \{ INV \land Post_r (x) \}
\]

\[
\{ Pre_r (atarg) \} \quad e.r (a) \quad \{ Post_r (atarg) \}
\]

Hoare-style “sequential” reasoning applies to synchronous and asynchronous calls

Targettable expressions are:
- Attached (statically known to be non-void)
- Handled by processor locked in current context

Targettability known statically from type
Example: asynchronous calls

\textit{store\_two (buf: separate BUFFER [INTEGER]; i, j: INTEGER)}

\textbf{require}

\begin{verbatim}
buf.count \leq buf.capacity - 2
\end{verbatim}

\textbf{do}

\begin{verbatim}
{buf.count \leq buf.capacity - 2}
buf.put (i)
{buf.count = old buf.count + 1 \land 
  buf.count \leq buf.capacity - 1}
buf.put (j)
{buf.count = old buf.count + 2 \land 
  buf.count \leq buf.capacity}
\end{verbatim}

\textbf{ensure}

\begin{verbatim}
buf.count = old buf.count + 2
\end{verbatim}

\textbf{end}
Implementation: two-level architecture

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

- POSIX threads
- .NET Threading
- ...
SCOOPLI: Library for SCOOP

Library-based solution

Implemented in Eiffel

Preprocessor and type checker
Elevator example architecture

For maximal concurrency, all objects are separate
Class BUTTON

class

    BUTTON

feature

    target: INTEGER

end
Class **CABIN_BUTTON**

class **CABIN_BUTTON** inherit **BUTTON**

*feature*

  *cabin: separate ELEVATOR*

  *request*

    -- Send to associated elevator a request to stop on level *target*.

    do

      *actual_request (cabin)*

    end

*actual_request (e: separate ELEVATOR)*

  -- Get hold of *e* and send a request to stop on level *target*.

  do

    *e.accept (target)*

  end

end
Class **ELEVATOR**

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*

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

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

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: separate ELEVATOR)
  -- Report that elevator e stopped on level position.
  do e.record_stop (position) end

feature {NONE}

  cabin: separate 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
- SCOOPLI 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.
Current developments & open problems

Semantic specification
Enriched type system
Wait on first of several events

Distribution and web services
Support for transactions
Deadlock prevention and detection
Extensions for real-time
Integration with compiler
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
- We can bring concurrent programming to the same level of safety and elegance as traditional programming
- We don’t really have a choice

SCOOP is here today, try it!

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