Concurrent Object-Oriented Programming

Prof. Dr. Bertrand Meyer

Lecture 2: an overview of SCOOP
my_queue: BUFFER[T]
...
if not my_queue.is_full then
  put(my_queue, t)
end
The issue

Concurrency everywhere:
- Multithreading
- Multitasking
- Networking, Web services, Internet
- Multicore

Can we bring concurrent programming to the same level of abstraction and convenience as sequential programming?
## Previous advances in programming

<table>
<thead>
<tr>
<th></th>
<th>“Structured programming”</th>
<th>“Object technology”</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

<table>
<thead>
<tr>
<th><strong>Sequential programming:</strong></th>
<th><strong>Concurrent programming:</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Used to be messy</td>
<td>Used to be messy</td>
</tr>
<tr>
<td>Still hard but key improvements:</td>
<td><strong>Still messy</strong></td>
</tr>
<tr>
<td>- Structured programming</td>
<td>Example: threading models in most popular approaches</td>
</tr>
<tr>
<td>- Data abstraction &amp; object technology</td>
<td></td>
</tr>
<tr>
<td>- Design by Contract</td>
<td>Development level: sixties/seventies</td>
</tr>
<tr>
<td>- Genericity, multiple inheritance</td>
<td></td>
</tr>
<tr>
<td>- Architectural techniques</td>
<td>Only understandable through operational reasoning</td>
</tr>
</tbody>
</table>
The chasm

Theoretical models, process calculi... Elegant theoretical basis, but

- Little connection with practice (some exceptions, e.g. BPEL)
- Handle concurrency aspects only

Practice of concurrent & multithreaded programming

- Little influenced by above
- Low-level, e.g. semaphores
- Poorly connected with rest of programming model
Wrong (in my opinion) assumptions

“Objects are naturally concurrent” (Milner)

- Many attempts, often based on “Active objects” (a self-contradictory notion)
- Lead to artificial issue of “Inheritance anomaly”

“Concurrency is the basic scheme, sequential programming a special case” (many)

- Correct in principle, but in practice we understand sequential best
Simple Concurrent Object-Oriented Programming

Evolved through last decade; *CACM* (1993) and chap. 32 of *Object-Oriented Software Construction, 2nd edition*, 1997

Implemented at ETH, ongoing integration into EiffelStudio

Current state is described in Piotr Nienaltowski’s 2007 ETH PhD dissertation
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
Typical traditional code

Listing 4.33: Variables for Tanenbaum’s solution

1  state = ['thinking'] * 5
2  sem = [Semaphore(0) for i in range(5)]
3  mutex = Semaphore(1)

The initial value of state is a list of 5 copies of ‘thinking’. sem is a list of 5 semaphores with the initial value 0. Here is the code:

Listing 4.34: Tanenbaum’s solution

1  def get_fork(i):
2      mutex.wait()
3      state[i] = 'hungry'
4      test(i)
5      mutex.signal()
6      sem[i].wait()
7
8  def put_fork(i):
9      mutex.wait()
10     state[i] = 'thinking'
11     test(right(i))
12     test(left(i))
13     mutex.signal()
14
15  def test(i):
16     if state[i] == 'hungry' and
17     state (left (i)) != 'eating' and
18     state (right (i)) != 'eating':
19         state[i] = 'eating'
20     sem[i].signal()
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
put \( (b \in \text{BUFFER}[G]; v \in G) \)

-- Store \( v \) into \( b \).

require

not \( b\text{.is_full} \)

do...

ensure

not \( b\text{.is_empty} \)

end

\[
\text{my\_queue: BUFFER}[T]\
\]

\[
\text{if not my\_queue\_is\_full then}\
\]

\[
\text{put}\ (\text{my\_queue}, t)\
\]

end
Reasoning about objects: sequential

\{\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\}

Priming represents actual-formal argument substitution

Only $n$ proofs if $n$ exported routines!
In a concurrent context

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

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

No overlapping!
SCOOP rules

- One processor per object: “handler”
- At most one feature (operation) active on an object at any time
Feature call: sequential

\[ x.r(a) \]

Client

previous

\[ x.r(a) \]

next

Supplier

\[ r(x: A) \]

do

... 

end

Processor
Feature call: asynchronous

Client

previous

\(x \cdot r(a)\)

next

Client's handler

Supplier

\(r(x : A)\)

do

...  

end

Supplier's handler
The fundamental difference

To wait or not to wait:

- If same processor, synchronous
- If different processor, asynchronous

Difference must be captured by syntax:

- \( x: X \)
- \( x: \text{separate } X \quad -- \text{potentially different processor} \)

Fundamental semantic rule: \( x.r(a) \) waits for non-separate \( x \), doesn’t wait for separate \( x \).
Consistency rules: avoiding traitors

\[
\text{nonsep: } T \\
\text{sep: separate } T \\
\text{nonsep := sep} \\
\text{nonsep.p(a)}
\]
Trusting what you read

\[ my\textunderscore stack: \texttt{separate STACK[T]} \]

...\[ my\textunderscore stack.push(a) \]

... Instructions not affecting the stack...

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

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

```
push (stack: separate STACK [T]; value: T)
-- Add value on top of stack.
  do
    stack.push (value)
  end
```
Target of a separate call must be formal argument of enclosing routine:

\[
\text{put}(\text{buffer} : \text{separate} \text{ BUFFER [ } T \text{ ]; value : } T) \\
\quad \text{-- Store value into buffer.}
\]

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

To use separate object:

\[
\text{my_buffer: separate } \text{ BUFFER [ INTEGER ]} \\
\text{create my_buffer} \\
\text{put(my_buffer, 10)}
\]
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
Wait rule

A routine call with separate arguments will execute when all corresponding processors are available and hold them exclusively for the duration of the routine.
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
Resynchronization

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

The client will wait only when it needs to:

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

Lazy wait (Denis Caromel, wait by necessity)
Contracts

\[\text{put}(\text{buf}: \text{BUFFER}[\text{INTEGER}]; \ v: \text{INTEGER})\]

-- Store \(v\) into buffer.

\[\begin{align*}
\text{require} & \quad \text{not } \text{buf}.\text{is_full} \\
\text{do} & \quad \text{buf}.\text{put}(v) \\
\text{ensure} & \quad \text{not } \text{buf}.\text{is_empty} \\
\text{end}
\end{align*}\]

... put(my_buffer, 10)
my_queue: BUFFER [T]
...
if not my_queue.is_full then
    put (my_queue, t)
end
put(buf: BUFFER [INTEGER]; v: INTEGER)
   -- Store v into buffer.
   require
   not buf.is_full
   v > 0
   do
      buf.put(v)
   ensure
      not buf.is_empty
   end

... put(my_buffer, 10)
A call with separate arguments waits until:
- The corresponding objects are all available
- Preconditions hold

\( x.f(a) \)

where \( a \) is separate
Which semantics applies?

```
put (buf: separate BUFFER [INTEGER]; i: INTEGER)
  require
    not buf.is_full
    i > 0
  do
    buf.put (i)
  end

my_buffer: separate BUFFER [INTEGER]
put (my_buffer, 10)
```
Generalized semantics of preconditions

- Sequentiality is a special case of concurrency.
- Wait semantics always applies.
- Wait semantics boils down to correctness semantics for non-separate preconditions.
  - Smart compiler can detect some cases
  - Other cases detected at run time

Distinction between controlled and uncontrolled rather than separate and non-separate.
What about postconditions?

zurich, palaiseau: separate LOCATION

spawn_two_activities(loc1, loc2: separate LOCATION)

do
  loc1.do_job
  loc2.do_job
ensure
  loc1.is_ready
  loc2.is_ready
end

spawn_two_activities(zurich, palo_alto)
do_local_stuff
get_result(zurich)

Should we wait for zurich.is_ready?
Reasoning about objects: sequential

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

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

Only \(n\) proofs if \(n\) exported routines!
Refined proof rule (partial correctness)

\[
\{ \text{INV} \land \text{Pre}_r(x) \} \text{ body}_r, \{ \text{INV} \land \text{Post}_r(x) \}
\]

\[
\{ \text{Pre}_r(a^{cont}) \} \text{ e.r}(a) \{ \text{Post}_r(a^{cont}) \}
\]

Hoare-style sequential reasoning

Controlled expressions (known statically as part of the type system) are:

- Attached (statically known to be non-void)
- Handled by processor locked in current context
Elevator example architecture

For maximal concurrency, all objects are separate
What if a separate call, e.g. in

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

causes an exception?
Implementation: two-level architecture

Adaptable to many environments

Currently implemented for native Windows (using POSIX threads) and .NET
Status

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

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

Implementation: integrating into EiffelStudio
Performance evaluation

Theory:
- Deadlock prevention and detection
- Less restrictive model (see STM)
- Transactions
- Full-fledged semantics
- Distributed SCOOP, Web Services
Why SCOOP?

- Simple (one new keyword) yet powerful
- Easier and safer than common concurrent techniques, e.g. Java Threads
- Full concurrency support
- Full use of O-O and Design by Contract
- Retains ordinary thought patterns, modeling power of O-O
- Supports wide range of platforms and concurrency architectures
- Programmers need to sleep better!