Concepts of Concurrent Computation

Bertrand Meyer
Sebastian Nanz

Lecture 6: SCOOP principles
put \( b: \text{BUFFER}[G]; v: G \)

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

require

not \( b.is\_full \)

do

... 

do 

ensure

not \( b.is\_empty \)

end

my_queue: \text{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>Feature</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

**Sequential programming:**

Used to be messy

Still hard but key improvements:

- Structured programming
- Data abstraction & object technology
- Design by Contract
- Genericity, multiple inheritance
- Architectural techniques

**Concurrent programming:**

Used to be messy

*Still messy*

Example: threading models in most popular approaches

Development level: sixties/seventies

Only understandable through operational reasoning
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

Most up-to-date description until recently is Piotr Nienaltowski’s 2007 ETH PhD dissertation
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
Typical traditional code

Listing 4.33: Variables for Tanenbaum’s solution

```python
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

```python
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 \( b : BUFFER [G] ; v : G \)  
--- 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

put (my_queue, \( t \))

end
Reasoning about objects: sequential

{INV and $Pre_r$} body$_r$ \{INV and $Post_r$\}

{Pre$_r$} $x \cdot r \ (a)$ \{Post$_r$\}

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

Priming represents actual-formal argument substitution
In a concurrent context

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

Client 1  Client 2  Client 3

$r1$  $r2$  $r3$

No overlapping!

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

$$\{\text{Pre}_r', x.r(a) \text{ } \{\text{Post}_r'\}$$
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.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:

- \( x: T \)
- \( x: \text{separate } T \) -- Potentially different processor

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

\[\text{nonsep} : T\]

\[\text{sep} : \text{separate } T\]

\[\text{nonsep} := \text{sep}\]

\[\text{nonsep}.p(a)\]
Trusting what you read

\[ x, y : T \]
\[ my\_buffer: \text{separate} \, \text{QUEUE} [T] \]

\[ x := my\_buffer.item \]

\[ my\_buffer.put(a) \]

\[ y := my\_buffer.item \]

... Instructions not affecting the buffer...
Access control policy

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

\[
\text{put} \ (b: \text{separate} \ \text{QUEUE} \ [T]; \ \text{value}: \ T)
\]

--- Add \text{value}, FIFO-style, to \(b\).

\[
\begin{align*}
\text{do} \\
\quad b.\text{put} \ (\text{value}) \\
\text{end}
\end{align*}
\]
Target of a separate call must be formal argument of enclosing routine:

```plaintext
put (b: separate QUEUE[T]; value : T)
-- Store value into buffer.
```

```
do
   b.put (value)
end
```

To use separate object:

```plaintext
my_buffer: separate QUEUE[INTEGER]
create my_buffer
put (my_buffer, 10)
```
Separate argument rule

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

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

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

\[
value := x.some\_query
\]

Lazy wait (Denis Caromel, wait by necessity)
put (buf : separate QUEUE[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 )
\textbf{put (b: BUFFER [G]; v: G)}

-- Store v into b.

\textbf{require}

\textbf{not} \ b.is\_full

\textbf{do}

... \\

\textbf{ensure}

\textbf{not} \ b.is\_empty

\textbf{end}

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

...

if not \textbf{my\_queue.is\_full} then

\textbf{put (my\_queue, t )}

\textbf{end}
The `put` function in the `separate QUEUE [INTEGER]` context is defined as follows:

```plaintext
put (buf : separate QUEUE [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 )
```

The precondition of the put function becomes a wait condition as follows:

- **Precondition becomes wait condition**
Full synchronization rule

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

"Separate call":

\[ x.f(a) \] -- where \( a \) is separate
Which semantics applies?

$$\text{put} \left( \text{buf} : \text{separate} \ \text{QUEUE} [\text{INTEGER}] ; \ i : \ \text{INTEGER} \right)$$

\begin{align*}
\text{require} \\
& \quad \textbf{not} \ \text{buf.is_full} \\
& \quad i > 0 \\
\text{do} \\
& \quad \text{buf.put} \left( i \right) \\
\text{end}
\end{align*}

Wait condition

Correctness condition

$$\text{my_buffer} : \text{separate} \ \text{QUEUE} [\text{INTEGER}]$$

$$\text{put} \left( \text{my_buffer}, 10 \right)$$
Generalized semantics of preconditions

- Sequentiability 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, lausanne : separate LOCATION*

```plaintext
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, lausanne)
do_local_stuff
get_result (zurich)

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

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

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

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

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

\begin{align*}
\{\text{Pre}_r(a^\text{cont})\} & \text{ e.r}(a) \{\text{Post}_r(a^\text{cont})\}
\end{align*}

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.
Other aspects

What if a separate call, e.g. in

\[
\begin{array}{l}
  r(a: \text{separate } T) \\
  \hspace{1em} \text{do} \\
  \hspace{2em} a.f \\
  \hspace{2em} a.g \\
  \hspace{2em} a.h \\
  \hspace{1em} \text{end}
\end{array}
\]

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
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!