Robotics Programming Laboratory

Bertrand Meyer
Jiwon Shin

Lecture 3:
Introduction to concurrency & SCOOP
The SCOOP programming model

Basic operation of OO programming: \( x \cdot f (...) \)
Can be a command or a query:

\[
\text{r (c: separate CONFERENCE ; p: PAPER)}
\]

\[
\text{require c.submission_open}
\]

\[
\text{do}
\]

\[
\text{c.submit (p)}
\]

\[
\text{...}
\]

\[
\text{if c.accepted (p) then rejoice end}
\]

\[
\text{end}
\]

\[
\text{r (icse , latest)}
\]

-- Exclusive access

-- Exclusive access when needed

-- Synchronous

-- Asynchronous

-- Waiting
Four risks

Data race
- Incorrect concurrent access to shared data

Deadlock
- Computation cannot progress because of circular waiting

Starvation
- Execution favors certain processes over others, which never get executed

Priority inversion
- Locks cause a violation of priority rules
Thank you for calling Ecstatic Opera Company. How can I help you?

(Joan) I need a single seat for next Tuesday’s performance of *Pique Dame*.

Let me check... You’re in luck! Just one left. Eighty dollars.

Great. I’ll go for it.

*Just a moment while I book it.*

Thanks.

*Sorry, there are no more seats available for Tuesday.*
## Data race: scenario

<table>
<thead>
<tr>
<th>Time step</th>
<th>Active participant</th>
<th>Request or action</th>
<th>Answer or result</th>
<th>Available seats</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Theatre</td>
<td>Available seats?</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Jane</td>
<td>Seats left?</td>
<td>Yes</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>Joan</td>
<td>Seats left?</td>
<td>Yes</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>Joan (fast to react)</td>
<td>Please book!</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>Jane (slow to react)</td>
<td>Please book!</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>Jane’s agent (fast to act)</td>
<td>Try to book</td>
<td>Success</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>Joan’s agent (slow to act)</td>
<td>Try to book</td>
<td>Failure</td>
<td>0</td>
</tr>
</tbody>
</table>
(Jane)

- I’d like to change my Tuesday evening seat for the matinee performance.
- *Both shows are sold out, but I heard there was a customer who wanted to change the other way around. Matinee booking is handled by a different office, so let me call them and make the change.*
- Thanks.
- *(Ten minutes later.*) "The number is still busy."
## Deadlock: scenario

<table>
<thead>
<tr>
<th>Time step</th>
<th>Active participant</th>
<th>Request or action</th>
<th>Answer or result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Agent 1</td>
<td>Matinee available for exchange?</td>
<td>Yes</td>
</tr>
<tr>
<td>2</td>
<td>Agent 2</td>
<td>Evening available for exchange?</td>
<td>Yes</td>
</tr>
<tr>
<td>3</td>
<td>Agent 1</td>
<td>Start dialing call to agent 2</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Agent 2</td>
<td>Start dialing call to agent 1</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Agent 1</td>
<td>Finish dialing</td>
<td>Busy signal, because agent 2 is trying to call</td>
</tr>
<tr>
<td>6</td>
<td>Agent 2</td>
<td>Finish dialing</td>
<td>Busy signal, because agent 1 is trying to call</td>
</tr>
<tr>
<td>7</td>
<td>Agent 1 &amp; Agent 2</td>
<td>Repeat steps 3 to 6 forever as the result remains the same: busy signals</td>
<td></td>
</tr>
</tbody>
</table>
Starvation

Jane keeps calling, but agents always pick up someone else's call
Execution sequences

<table>
<thead>
<tr>
<th></th>
<th>P1</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>x := 0</td>
<td>x := 2</td>
</tr>
<tr>
<td>2</td>
<td>x := x + 1</td>
<td>x := 2</td>
</tr>
</tbody>
</table>

- Execution can give rise to this execution sequence:

<table>
<thead>
<tr>
<th>Thread ID</th>
<th>Instruction</th>
<th>Variable Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1 1</td>
<td>x := 0</td>
<td>x := 0</td>
</tr>
<tr>
<td>P2 1</td>
<td>x := 2</td>
<td>x := 2</td>
</tr>
<tr>
<td>P1 2</td>
<td>x := x + 1</td>
<td>x := 3</td>
</tr>
</tbody>
</table>

Instruction executed with Thread ID and line number

Variable values after execution of the code on the line
Execution sequences

<table>
<thead>
<tr>
<th>x := 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>

Possible execution sequences considering all interleavings:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>P2</td>
<td>1</td>
<td>x := 2</td>
</tr>
<tr>
<td></td>
<td>x = 2</td>
<td></td>
</tr>
<tr>
<td>P1</td>
<td>1</td>
<td>x := 0</td>
</tr>
<tr>
<td></td>
<td>x = 0</td>
<td></td>
</tr>
<tr>
<td>P1</td>
<td>2</td>
<td>x := x + 1</td>
</tr>
<tr>
<td>x = 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>P1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>x = 0</td>
<td></td>
</tr>
<tr>
<td>P1</td>
<td>2</td>
<td>x := x + 1</td>
</tr>
<tr>
<td>x = 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>P2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>x = 2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>P1</td>
<td>2</td>
</tr>
<tr>
<td>x = 3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Data races (race conditions)

If processes (OS processes, threads) are completely independent, concurrency is easy.

Usually, however, threads *interfere* with each other by accessing and modifying common resources, such as variables and objects:

- Unwanted dependency of the computation’s result on nondeterministic interleaving is a *race condition* or *data race*.
- Such errors can stay hidden for a long time and are difficult to find by testing.
Dining philosophers
The dining philosophers problem

$n$ philosophers are seated around a table; between each pair there is a single fork.
Each philosopher only thinks and eats.
To eat, a philosopher needs both left and right forks (so two adjacent philosophers cannot eat at the same time).

The problem: devise an algorithm enabling philosophers to follow this scheme, without deadlock.
Dining philosophers: solution attempt 1

Each philosopher first picks up the right fork, then the left fork, and then starts eating; after having eaten, the philosopher puts down the left fork, then the right one

- The philosophers can deadlock!
Dining philosophers: solution attempt 2

Each philosopher successively:

- Picks up right fork and the left fork \textit{at the same time}
- Starts eating
- After having eaten, puts them both back down

A philosopher could \textit{starve}!
To prove freedom from starvation or other liveness properties, *fairness assumptions* are sometimes needed.

Fairness is concerned with a fair resolution of nondeterminism.

- **Weak fairness**: if an action is continuously enabled, i.e. never temporarily disabled, then it has to be executed infinitely often.
- **Strong fairness**: if an action is infinitely often enabled, but not necessarily always, then it has to be executed infinitely often.
SCOOP background

Simple Concurrent Object-Oriented Programming

First version described in CACM article (1993) and chapter 32 of Object-Oriented Software Construction, 2nd edition, 1997

Prototype implementation at ETH (2005-2010)
Recent production implementation at Eiffel Software, part of EiffelStudio

Recent descriptions: Piotr Nienaltowski’s 2007 ETH PhD; Morandi, Nanz, Meyer (2011)
Example 1: bank transfer, from sequential to concurrent

```plaintext
transfer (source, target: separate ACCOUNT;
    amount: INTEGER)

-- Transfer amount, if available, from source to target.
do
    if source.balance >= amount then
        source.withdraw (amount)
        target.deposit (amount)
    end
end
```

<table>
<thead>
<tr>
<th></th>
<th>Jane</th>
<th>Jill</th>
<th>Joan</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>100</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>-100</td>
<td>0</td>
<td>100</td>
<td></td>
</tr>
</tbody>
</table>

transfer (Jane, Jill, 100)
transfer (Jane, Joan, 100)
Bank transfer (better version)

```plaintext
transfer (source, target: ACCOUNT; amount: INTEGER)
  -- Transfer amount from source to target.
require
  source.balance >= amount
do
  source.withdraw (amount)
  target.deposit     (amount)
ensure
  source.balance = old source.balance - amount
  target.balance = old target.balance + amount
end
```
Example 2: hexapod robot

Hind legs have force sensors on feet and retraction limit switches
Hexapod locomotion

Alternating protraction and retraction of tripod pairs

- Begin protraction only if partner legs are down
- Depress legs only if partner legs have retracted
- Begin retraction when partner legs are up

Ganesh Ramanathan, Benjamin Morandi, IROS 2011
Hexapod coordination rules

**R1**: Protraction can start only if partner group on ground

**R2.1**: Protraction starts on completion of retraction

**R2.2**: Retraction starts on completion of protraction

**R3**: Retraction can start only when partner group raised

**R4**: Protraction can end only when partner group retracted

Sequential implementation

```java
TripodLeg lead = tripodA;
TripodLeg lag = tripodB;

while (true)
{
    lead.Raise();
    lag.Rettract();
    lead.Swing();
    lead.Drop();

    TripodLeg temp = lead;
    lead = lag;
    lag = temp;
}
```
Multi-threaded implementation

```csharp
private object m_protractionLock = new object();

private void ThreadProcWalk(object obj)
{
    TripodLeg leg = obj as TripodLeg;
    while (Thread.CurrentThread.ThreadState != ThreadState.AbortRequested)
    {
        // Waiting for protraction lock
        lock (m_protractionLock)
        {
            // Waiting for partner leg drop
            leg.Partner.DroppedEvent.WaitOne();
            leg.Raise();
        }

        leg.Swing();

        // Waiting for partner retraction
        leg.Partner.RetractedEvent.WaitOne();
        leg.Drop();

        // Waiting for partner raise
        leg.Partner.RaisedEvent.WaitOne();
        leg.Retract();
    }
}
```
begin_protracion(partner, me:separate LEG_GROUP_SIGNALER)

   require
    my_legs_retracted : me.legs_retracted
    partner_down : partner.legs_down
    partner_not_protracting : not partner.protracion_pending

   do
      io.put_string (group_name)
      io.put_string (" : begin_protracion ")
      io.put_new_line

      tripod.lift

      me.set_protracion_pending(true)
   end
Hexapod coordination rules

R1: Protraction can start only if partner group on ground
R2.1: Protraction starts on completion of retraction
R2.2: Retraction starts on completion of protraction
R3: Retraction can start only when partner group raised
R4: Protraction can end only when partner group retracted

Example 3: dining philosophers

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()
```
class PHILOSOPHER feature
  live
  do
    from getup until over loop
      think ;
      eat (left, right)
    end
  end
end

eat (l, r: separate FORK)
  -- Eat, having grabbed l and r.
  do ... end

getup do ... end
over: BOOLEAN

end
Example 4: elevator system

From: *Object-Oriented Software Construction*

For maximal concurrency, all objects are separate
The design of SCOOP (and this presentation)

To achieve the preceding goals, SCOOP makes a number of restrictions on the concurrent programming model.

This presentation explains and justifies these restrictions one after the other.

The goal is not to limit programmers but to enable them to reason about the programmers.
The design of SCOOP

SCOOP intends to make concurrent programming as predictable as sequential programming.

A key criterion is “reasonability” (not a real word!): the programmer’s ability to reason about the execution of programs based only on their text.

- As in sequential O-O programming, with contracts etc.

SCOOP is not a complete rework of basic programming schemes, but an incremental addition to the basic O-O scheme: one new keyword

- “Concurrency Made Easy”
Handling concurrency simply

SCOOP narrows down the distinction between sequential & concurrent programming to six properties, studied next:

- **(A)** Single vs multiple “processors”
- **(B)** Regions
- **(C)** Synchronous vs asynchronous calls
- **(D)** Semantics of argument passing
- **(E)** Semantics of resynchronization (lazy wait)
- **(F)** Semantics of preconditions
The starting point (A): processors

To perform a computation is
- To apply certain actions
- To certain objects
- Using certain processors

Sequential: one processor
Concurrent: any number of 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) ...

The SCOOP model is abstract and does not specify the mapping to such actual computational resources
Object-oriented programming

The key operation is “feature call”

\[ x \cdot f(\text{args}) \]

where \( x \), the \textbf{target} of the call, denotes an object to which the call will apply the feature \( f \)

Which processor is in charge of executing such a call?
All calls targeting a given object will be executed by a single processor

- The set of objects handled by a given processor is called a *region*
- The processor in charge of an object is its *handler*
A consequence: regions

All calls targeting a given object will be executed by a single processor

- The set of objects handled by a given processor is called a *region*
SCOOP restriction: one handler per object

- One processor per object: “handler”

- At most one feature (operation) active on an object at any time
The notion of handler implies a partitioning of the set of objects:

- The set of objects handled by a given processor is called a **region**
- Handler rule implies one-to-one correspondence between processors and regions
(C) The sequential view: O-O feature calls

\[ x.r(a) \]

Client

previous

next

Processor

Supplier

\[ r(x : A) \]

do

...  

end
(C) The concurrent form of call: asynchronous

Client

\[ \text{previous} \]
\[ x.r(a) \]
\[ \text{next} \]

Client’s handler

Supplier

\[ r(x:A) \]
\[ \text{do} \]
\[ \ldots \]
\[ \text{end} \]

Supplier’s handler
The two forms of O-O call

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: a call \( x.r(a) \)
- Waits (i.e. is synchronous) for non-separate \( x \)
- Does not wait (is asynchronous) for separate \( x \)
Why potentially separate?

**separate** declaration only states that the object *might* be handled by a different processor

- In class A: \( x: \text{separate } B \)
- In class B: \( y: \text{separate } A \)
- In A, what is the type of \( x \cdot y \)?

In some execution, the value might be a reference to an object in the current region
Call vs application

With asynchrony we must distinguish between feature call and feature application.

The execution

\[ x \cdot r (...) \]

is the call, and (with \( x \) separate) will not wait (the client just logs the call).

The execution of \( r \) happens later and is called the feature application.
Consistency rules: avoiding traitors

\[\text{nonsep}: T\]

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

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

\[\text{nonsep}.p(a)\]

Traitor!

More traitor protection through the type system!
(D) Access control policy

Since separate calls are asynchronous there is a real danger of confusion

Consider for example

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

\[
... \\
\text{remote\_stack.put(a)} \\
... \text{Instructions not affecting the stack...} \\
y := my\_stack.item
\]
(D) Access control policy

SCOOP requires the target of a separate call to be a **formal argument** of enclosing routine:

\[ put(s: \text{separate}\ STACK[T]; value: T) \]

\[ \text{-- Store value into } s. \]

\[ \text{do} \]

\[ s.put(value) \]

\[ \text{end} \]

To use separate object:

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

create \[ my\_stack \]

\[ \text{put}(my\_stack, 10) \]
The target of a separate call must be an argument of the enclosing routine.

Separate call: \( x \cdot f(...) \) where \( x \) is separate.
A routine call guarantees exclusive access to the handlers (the processors) of all separate arguments

\[ a\_routine (\text{nonsep}_a, \text{nonsep}_b, \text{sep}_c, \text{sep}_d, \text{sep}_e) \]

Exclusive access to \[ \text{sep}_c, \text{sep}_d, \text{sep}_e \] within \[ a\_routine \]
transfer (source, target: \textbf{separate} ACCOUNT; 
amount: INTEGER)

\begin{verbatim}
-- Transfer amount, if available, from source to target.
do
  if source.balance >= amount then
    source.withdraw (amount)
    target.deposit (amount)
  end
end
\end{verbatim}
class PHILOSOPHER feature
    live
    do
        from getup until over loop
            think;
            eat(left, right)
        end
    end
end

eat(l, r: separate FORK)
    -- Eat, having grabbed l and r.
    do ... end

getup do ... end
over: BOOLEAN
end
(D) What the wait rule means

Beat enemy number one in concurrent world: atomicity violations

- Data races
- Illegal interleaving of calls

Data races cannot occur in SCOOP
A routine call guarantees exclusive access to the handlers (the processors) of all separate arguments

\[ \text{a\_routine}(\text{nonsep\_a}, \text{nonsep\_b}, \text{sep\_c, sep\_d, sep\_e}) \]

Exclusive access to \[\text{sep\_c, sep\_d, sep\_e}\] within \text{a\_routine}
Semantics vs implementation

Older SCOOP literature says that feature application “waits” until all the separate arguments’ handlers are available. This is not necessary! What matters is exclusive access: implementation does not have to wait unless semantically necessary. The implementation performs some of these optimizations.

\[
f(a, b, c : \text{separate } T) \\
do \\
\quad \text{something}_\text{else} \\
a.r \\
b.s \\
end
\]

No need to wait for \(a\) and \(b\) until here
No need to wait for \(c\)!
How do we resynchronize after asynchronous (separate) call?

No explicit mechanism!

The client will wait when, and only when, it needs to:

\[
x.f \\
x.g(a) \\
y.f \\
... \\
value := x.some_query
\]

Lazy wait (also known as wait by necessity)
(E) Synchrony vs asynchrony revisited

For a separate target $x$:

- $x \cdot \text{command}(...) \text{ is asynchronous}$
- $v := x \cdot \text{query}(...) \text{ is synchronous}$
Exercise

If we do want to resynchronize explicitly, what do we do?
What becomes of contracts, in particular preconditions, in a concurrent context?
**put**

```plaintext
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  

```plaintext
put(my_queue, t)
```

end
**Contracts**

```plaintext
put (b: separate QUEUE [INTEGER]; v: INTEGER)
    -- Store v into b.
require
    not b.is_full
    v > 0
do
    b.put(v)
ensure
    not b.is_empty
end

... put (my_buffer, 10)
```
**Contracts**

\[ \text{put}(b: \text{BUFFER}[\text{INTEGER}]; i: \text{INTEGER}) \]

-- Store \( i \) into buffer.

\[
\text{require} \\
\quad \text{not } b.\text{is_full} \\
\quad i > 0 \\
\text{do} \\
\quad b.\text{put}(i) \\
\text{ensure} \\
\quad \text{not } b.\text{is_empty} \\
\text{end}
\]

Precondition becomes wait condition

\[ \text{put}(\text{my_buffer}, 10) \]
Bank transfer (version with contracts)

```plaintext
transfer (source, target: ACCOUNT; amount: INTEGER)
    -- Transfer amount from source to target.
    require
        source.balance >= amount
    do
        source.withdraw (amount)
        target.deposit     (amount)
    ensure
        source.balance = old source.balance - amount
        target.balance = old target.balance + amount
    end
```
(F) Full synchronization rule

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

"Separate call":

\[ x.f(a) \quad -- \text{where } a \text{ is separate} \]
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

The different semantics is surprising at first:
- Separate: wait condition
- Non-separate: correctness condition

At a high abstraction level, however, we may consider that
- Wait semantics always applies in principle
- Sequentiality is a special case of concurrency
- Wait semantics boils down to correctness semantics for non-separate preconditions.
  - Smart compiler can detect some cases
  - Other cases detected at run time