ACM Highlights

• Learning Center tools for professional development: [http://learning.acm.org](http://learning.acm.org)
  • The Safari Learning Platform featuring the **entire Safari collection of nearly 50,000** technical books, video courses, O’Reilly conference videos, learning paths, tutorials, case studies
  • 1,800+ Skillsoft courses, 4,800+ online books, and 30,000+ task-based short videos for software professionals covering programming, data management, DevOps, cybersecurity, networking, project management, and more; including training toward top vendor certifications such as AWS, CEH, Cisco, CISSP, CompTIA, Oracle, RedHat, PMI.
  • 1,200+ books from Elsevier on the ScienceDirect platform (including Morgan Kaufmann and Syngress titles)
• Learning Webinars from thought leaders and top practitioners
• Podcast interviews with innovators, entrepreneurs, and award winners
• Popular publications:
  • Flagship *Communications of the ACM (CACM)* magazine: [http://cacm.acm.org](http://cacm.acm.org)
  • *ACM Queue* magazine for practitioners: [http://queue.acm.org](http://queue.acm.org)
• The **ACM Code of Ethics**, a set of principles and guidelines designed to help computing professionals make ethically responsible decisions in professional practice: [https://ethics.acm.org](https://ethics.acm.org)
• ACM Digital Library, the world’s most comprehensive database of computing literature: [http://dl.acm.org](http://dl.acm.org)
• International conferences that draw leading experts on a broad spectrum of computing topics: [http://www.acm.org/conferences](http://www.acm.org/conferences)
• Prestigious awards, including the ACM A.M. Turing and ACM Prize in Computing: [http://awards.acm.org](http://awards.acm.org)
• And much more… [http://www.acm.org](http://www.acm.org).
Concurrency Made Easy:
Concurrent object-oriented programming
with SCOOP

ACM Webinar, 15 November 2018
Bertrand Meyer

Material copyright Bertrand Meyer, 2018
• Learning Center tools for professional development: http://learning.acm.org
  • The Safari Learning Platform featuring the entire Safari collection of nearly 50,000 technical books, video courses, O’Reilly conference videos, learning paths, tutorials, case studies
  • 1,800+ Skillsoft courses, 4,800+ online books, and 30,000+ task-based short videos for software professionals covering programming, data management, DevOps, cybersecurity, networking, project management, and more; including training toward top vendor certifications such as AWS, CEH, Cisco, CISSP, CompTIA, Oracle, RedHat, PMI.
  • 1,200+ books from Elsevier on the ScienceDirect platform (including Morgan Kaufmann and Syngress titles)
  • Learning Webinars from thought leaders and top practitioners
  • Podcast interviews with innovators, entrepreneurs, and award winners
• Popular publications:
  • Flagship Communications of the ACM (CACM) magazine: http://cacm.acm.org
  • ACM Queue magazine for practitioners: http://queue.acm.org
• The ACM Code of Ethics, a set of principles and guidelines designed to help computing professionals make ethically responsible decisions in professional practice: https://ethics.acm.org
• ACM Digital Library, the world’s most comprehensive database of computing literature: http://dl.acm.org
• International conferences that draw leading experts on a broad spectrum of computing topics: http://www.acm.org/conferences
• Prestigious awards, including the ACM A.M. Turing and ACM Prize in Computing: http://awards.acm.org
• And much more... http://www.acm.org.
ACM Highlights

- Learning Center tools for professional development: [http://learning.acm.org](http://learning.acm.org)
  - The Safari Learning Platform featuring the **entire Safari collection of nearly 50,000** technical books, video courses, O’Reilly conference videos, learning paths, tutorials, case studies
  - 1,800+ Skillsoft courses, 4,800+ online books, and 30,000+ task-based short videos for software professionals covering programming, data management, DevOps, cybersecurity, networking, project management, and more; including training toward top vendor certifications such as AWS, CEH, Cisco, CISSP, CompTIA, Oracle, RedHat, PMI.
  - 1,200+ books from Elsevier on the ScienceDirect platform (including Morgan Kaufmann and Syngress titles)
  - Learning Webinars from thought leaders and top practitioners
  - Podcast interviews with innovators, entrepreneurs, and award winners
- Popular publications:
  - Flagship *Communications of the ACM (CACM)* magazine: [http://cacm.acm.org](http://cacm.acm.org)
  - *ACM Queue* magazine for practitioners: [http://queue.acm.org](http://queue.acm.org)
- The **ACM Code of Ethics**, a set of principles and guidelines designed to help computing professionals make ethically responsible decisions in professional practice: [https://ethics.acm.org](https://ethics.acm.org)
- ACM Digital Library, the world’s most comprehensive database of computing literature: [http://dl.acm.org](http://dl.acm.org)
- International conferences that draw leading experts on a broad spectrum of computing topics: [http://www.acm.org/conferences](http://www.acm.org/conferences)
- Prestigious awards, including the ACM A.M. Turing and ACM Prize in Computing: [http://awards.acm.org](http://awards.acm.org)
- And much more... [http://www.acm.org](http://www.acm.org).
Talk Back

• Use Twitter widget to Tweet your favorite quotes from today’s presentation with hashtag #ACMLearning

• Submit questions and comments via Twitter to @acmeducation – we’re reading them!

• Use the sharing widget in the bottom panel to share this presentation with friends and colleagues.

• The ACM Discourse Page is available for post-webinar discussion – https://on.acm.org
Concurrency Made Easy: Concurrent object-oriented programming with SCOOP

ACM Webinar, 15 November 2018
Bertrand Meyer

Material copyright Bertrand Meyer, 2018
SCOOP

- Built-in guarantee of **no data races**
- Close connection to O-O modeling
- Natural use of O-O mechanisms such as inheritance
- Built-in fairness
- Removes many concerns from programmer
- Supports many different forms of concurrency
- Retains accepted patterns of reasoning about programs
- Simple to learn and use
SCOOP performance is competitive

(Smaller is better)

Over all benchmarks, SCOOP is the best performing of the data-race-free frameworks (Erlang/Haskell)
What is SCOOP?

Simple Concurrent Object-Oriented Programming

Incremental addition to basic O-O scheme: **one new keyword**

- Basic ideas go back to 1993 CACM paper
- Implementations at Eiffel Software (processes) then ETH Zurich and Eiffel Software (threads)
- “Concurrency Made Easy” Advanced Investigator Grant project from European Research Council, €2.5M, ETH then Politecnico di Milano
- Standard part of Eiffel language and IDE (EiffelStudio)
The goal

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

Ways to approach concurrency:

1. It’s the general setup, sequential is just a special case
2. We understand sequential, let’s keep close to it

“Reasonability”
The only foreseeable way to continue advancing performance is to match parallel hardware with parallel software. There has been genuine progress on the software front in specific fields, such as some scientific applications. Heroic programmers can exploit vast amounts of parallelism. However, none of those developments comes close to the ubiquitous support for programming parallel hardware required to ensure that IT’s effect on society over the next two decades will be as stunning as it has been over the last half-century.

The Future of Computing Performance: Game Over or Next Level? https://www.nap.edu/read/12980/chapter/2 (slightly abridged)
Dijkstra: *Goto considered harmful*, 1968*

Our intellectual powers are geared to visualize static relations, not processes *evolving in time*.

Hence we should strive to shorten the conceptual gap between the static program (in text space) and the dynamic process (in time).


*Slightly abridged*
Dining philosophers (Tanenbaum)

```python
def getfork(i):
    mutex.wait()
    state[i] = 'hungry'
    test(i)
    mutex.signal()
    sem[i].wait()

def putfork(i):
    mutex.wait()
    state[i] = 'thinking'
    test(right(i))
    test(left(i))
    mutex.signal()

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

    def test(i):
        if state[i] == 'hungry' and state[left(i)] != 'eating' and state[right(i)] != 'eating':
            state[i] = 'eating'
            sem[i].signal()
```
Dining philosophers in SCOOP

class PHILOSOPHER feature

live

do

from getup until is_over loop

think ; eat (left, right)

end

end

eat (l, r: separate FORK)

-- Eat, having grabbed l and r.

do ... l.pick ; ... r.pick ; ... end

getup do ... end

is_over: BOOLEAN

end
Bank transfer

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

Class invariant: balance >= 0
Bank transfer (simplified)

transfer (source, target: ACCOUNT; amount: INTEGER)

-- If enough funds, transfer amount from source to target.
do
    if source.balance >= amount then
        source.withdraw (amount)
        target.deposit (amount)
    end
end
Data race

transfer (source, target: ACCOUNT;
  amount: INTEGER)
  -- If enough funds, transfer amount from source to target.
do
  if source.balance >= amount then
    source.withdraw (amount)
    target.deposit  (amount)
  end
end

transfer (Jane, Jill, 100)
  transfer (Jane, Joan, 100)
The inability to reason from APIs

```plaintext
if acc1.balance >= 100
then transfer (acc1, acc2, 100) end

if acc1.balance >= 100
then transfer (acc1, acc3, 100) end
```

```
transfer (source, target: ACCOUNT; amount: INTEGER)
-- Transfer amount from source to target.
require
  source.balance >= amount
ensure
  source.balance = old source.balance – amount
  target.balance = old target.balance + amount
```

Invariant:

```
balance >= 0
```
Bank transfer in SCOOP

\[
\text{transfer (source, target: separate ACCOUNT; amount: INTEGER)}
\]

-- Transfer amount from source to target.

require

\[\text{source.balance} >= \text{amount}\]

do

\[\text{source.withdraw (amount)}\]
\[\text{target.deposit (amount)}\]

ensure

\[\text{source.balance} = \text{old source.balance} - \text{amount}\]
\[\text{target.balance} = \text{old target.balance} + \text{amount}\]

end
SCOOP choices

1. Object-oriented programming
2. Processors
3. Partitioning of the object space
4. Dynamic processor creation
5. Asynchronous command calls
6. Mutual exclusion on objects
7. Multiple simultaneous object reservation
8. Forced encapsulation
9. Resynchronization through queries
10. Conditional waiting through preconditions
Choice 1: object-oriented programming

- (Static) type and module structure: class
- (Dynamic) data structure: object
- Inheritance for (static) reuse and (dynamic) binding
Choice 2: processors

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

Can be implemented as:
- Computer CPU
- Process
- Thread

Will be mapped to computational resources
Choice 3: map object structure to processor structure

Fundamental operation in OO programming:

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

**Qualified call** “targeted” to \( x \)
(also: “message passing”),
All calls targeted to a given object are performed by a single processor, called the object’s handler.
Choice 4: asynchronous command calls

Qualified call in the sequential world:

Client

\[ \text{previous} \]

\[ \text{x}.r \,(v) \]

\[ \text{next} \]

Supplier

\[ r \,(u : T) \]

\[ \text{do} \]

\[ \text{u.something} \]

\[ \text{end} \]
Qualified calls in a concurrent world

Client

\[ x \cdot r(v) \]

next

Client’s handler

\[ r(u : T) \]

\[ \text{do} \]

\[ \text{u.something} \]

\[ \text{end} \]

Supplier

Supplier’s handler

Processors
Call versus application

The execution of a call requested by a processor on objects in another region is asynchronous.

We must distinguish between:

- Routine (method) call
- Routine application
Semantic guarantees

Rule 1 (causality): for given call, application occurs after call

Rule 2 (consistency): from given client processor to given supplier processor, application order is call order

No guarantee between > 2 processors
The two forms of O-O call

“Separate” means: possibly in a different region

A command call $x.r(a)$ is:

- Synchronous (waits) for non-separate $x$
- Asynchronous (does not wait) for separate $x$

Difference captured by syntax:

- $x: T$
- $x: separate T$ -- Potentially different region
class CLIENT feature

messages: LIST [STRING]
downloader: separate DOWNLOADERviewer: separate VIEWER

... end

Demo: an email system, from sequential to concurrent
Choice 5: dynamic processor creation

With

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

the creation instruction

\[ \text{create } x \]

creates an object (as usual), but also

- Creates a new region
- Puts the new object in that region
- Starts the associated processor
Choice 6: mutual exclusion on objects

At any given time, at most one operation in progress on any given object

(In fact, on objects in any given region)

No intra-object concurrency
Reasoning about objects, concurrently

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

\[
\begin{align*}
\text{Client 1} & \quad \text{Client 2} & \quad \text{Client 3} \\
r_1 & \quad r_2 & \quad r_3
\end{align*}
\]

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

\[
\{\text{Pre}_{r'}\} \quad x.r(a) \quad \{\text{Post}_{r'}\}
\]
Choice 7: multiple simultaneous object reservation

A call

\[ x_{ir} (a_1, a_2, ...) \]

will wait until it has been able to lock all the separate objects associated with the arguments \( a_1, a_2, ... \)

Guarantees mutual exclusion

Applies to locking *any number of objects*
Dining philosophers in SCOOP

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

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

getup do ... end
is_over: BOOLEAN
end
```
Another example of mutual exclusion

This routine will lock $b$:

\[
\text{put} \ (b : \text{separate} \ \text{QUEUE} \ [T] ; \ text{value :} \ T) \\
\text{-- Add value, FIFO-style, to } b. \\
\text{do} \\
\quad b.\text{put} \ (\text{value}) \\
\text{end}
\]

The buffer update $b.\text{put} \ (\text{value})$ is mutually exclusive and safe
Dining philosophers in SCOOP (slight variant)

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
Choice 8: forced encapsulation

Locking through argument passing is enforced in SCOOP:

The target of a separate call must be a formal argument of enclosing routine

Invalid code (compile-time error):

```
buff: separate QUEUE[T]
...
buff.put (value1)
buff.put (value2)
```
Making this code valid

insert (buff: separate QUEUE[T])
  do
    ...
    buff.put (value1)
    buff.put (value2)
  end
Limiting the proliferation of wrappers

You can also use the `separate` instruction:

```
buff: separate QUEUE [T]
...
separate buff as b do
  buff.put (value1)
  buff.put (value2)
end
```
Choice 9: resynchronize through queries

How do we resynchronize after asynchronous (separate) call?

\[
\begin{align*}
    & x.\text{command1} \ (u, \ v) \\
    & x.\text{command2} \ (a, \ b) \\
    & x.\text{command3} \\
    & \ldots \\
    & \text{value} := x.\text{query1}
\end{align*}
\]

Answer: the client will wait when, and only when, it needs to

Terminology:

- A command does not return a result (procedure).
- A query returns a result (function or attribute).
Lazy wait (or wait by necessity, Caromel)

A command call $x.c$ is asynchronous

A query call $y := x.q$ is synchronous
Choice 10: conditional waiting through preconditions

What becomes of contracts, in particular preconditions, in a concurrent context?

```
put (b: separate QUEUE [INTEGER] ; v: INTEGER)
  -- Insert v into buffer b.
require
  not b.is_full
do
  b.put (v)
ensure
  not b.is_empty
end
```

In a client:
```
buff: separate QUEUE [INTEGER]
if not buff.is_full then
  put (buff, 10)
end
```

Precondition becomes wait condition
Condition synchronization in SCOOP

Application of a routine only proceeds when separate preconditions satisfied
A precondition is separate if it involves a call to a separate target

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

Correctness condition (no wait semantics)
Precondition becomes wait condition
Example: bank transfer

```haskell
transfer (source, target: separate 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
```
Another example: hexapod robot

Roboscoop framework
Hexapod locomotion

Dürr, Schmitz, Cruse:  
*Behavior-based modeling of hexapod locomotion*

in *Arthropod Structure & Development*, 2004

**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
begin_protraction (partner, me: separate LEG_GROUP)

require
me.legs_retracted
partner.legs_down
not partner.protraction_pending

do
tripod.lift
me.set_protraction_pending

end

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
Multi-threaded implementation

private object m_protractionLock = new object();

private void ThreadProcWalk(object obj)
{
    TripodLeg leg = obj as TripodLeg;
    while (Thread.CurrentThread.ThreadState != ThreadState.AbandonRequested)
    {
        // Waiting for protraction lock
        lock (m_protractionLock)
        {
            // Waiting for partner leg drop
            leg.Partner.DroppedEvent.WaitOne();
            leg.Swing();
            leg.Raise();
            // Waiting for partner retraction
            leg.Partner.RetractedEvent.WaitOne();
            leg.Drop();
            // Waiting for partner raise
            leg.Partner.RaisedEvent.WaitOne();
            leg.Retract();
        }
    }
}
Traitors and the SCOOP type system

a, b: PERSON
x, y: separate PERSON

x := a
b := y

r (f: separate PERSON) do ... end

x \dot{=} r (a)

Traitor: variable declared as non-separate which, at run time, may become attached to a separate object.
The type system guarantees the absence of traitors.
Teamwork

(ETH, Eiffel Software, Politecnico)

Georgiana Caltais
Alexei Kolesnichenko
Alexander Kogtenkov
Benjamin Morandi
Sebastian Nanz
Piotr Nienaltowski
Chris Poskitt

Ganesh Ramanathan
Andrey Rusakov
Roman Schmocker
Mischael Schill
Jiwon Shin
Emmanuel Stapf
Scott West

Plus many colleagues:
Jonathan Ostroff, Phil Brooke, Richard Paige, Manfred Broy, Jay Misra, Denis Caromel...
Open issues

Model:
- Multiple readers

Implementation
- Better support for distribution
- Continuation of work on GPUs
- Deadlock analysis

Theory
- Full proof rule
SCOOP

- Built-in guarantee of no data races
- Close connection to O-O modeling
- Natural use of O-O mechanisms such as inheritance
- Built-in fairness
- Removes many concerns from programmer
- Supports many different forms of concurrency
- Retains accepted patterns of reasoning about programs
- Simple to learn and use

To learn more (and try SCOOP):

http://cme.ethz.ch
http://eiffel.org
ACM: The Learning Continues...

- Questions/comments about this webcast? learning@acm.org
- ACM Code of Ethics: https://ethics.acm.org
- ACM’s Discourse Page: http://on.acm.org
- ACM Learning Webinars (on-demand archive): http://webinar.acm.org
- ACM Learning Center: http://learning.acm.org
- ACM Queue: http://queue.acm.org