ACM Highlights

- 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 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.

Concurrency Made Easy: Concurrent object-oriented programming with SCOOP

ACM Webinar,15 November 2018

Bertrand Meyer

ACM Highlights

- 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 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
  - 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 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.

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

SCOOP

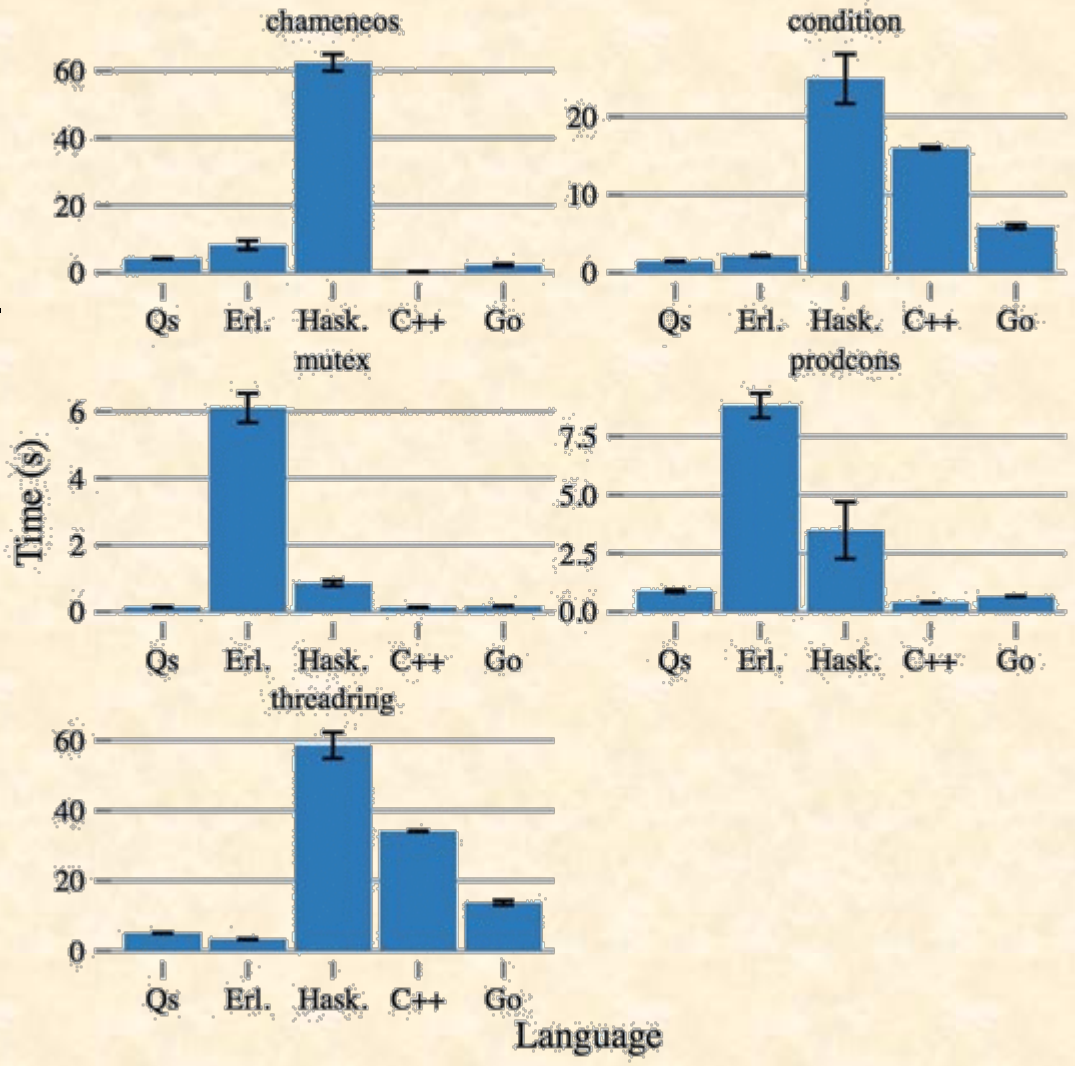
- Built-in guarantee of **no data races**
- Close connection to O-O modeling
- Natural use of O-O mechanisms such as inheritance
- Built-in fairness
- Removes many concerns from programmer
- Supports many different forms of concurrency
- Retains accepted patterns of reasoning about programs
- Simple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

SCOOP performance is competitive

Sebastian Nanz, Roman Schmocker
ESEC/FSE 2013

(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"

National Academies of Sciences, 2011

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).

Source: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.92.4846&rep=rep1&type=pdf

**Slightly abridged*

Dining philosophers (Tanenbaum)

```
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()
```

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

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

Source: https://www.cs.grinnell.edu/~weinman/courses/CSC213/2012F/labs/philosophers.html

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
```

Jane	Jill	Joan
100	0	0
0	100	0
-100	100	100

transfer (Jane, Jill, 100) 1

transfer (Jane, Joan, 100) 2

The inability to reason from APIs

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

```
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
```

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

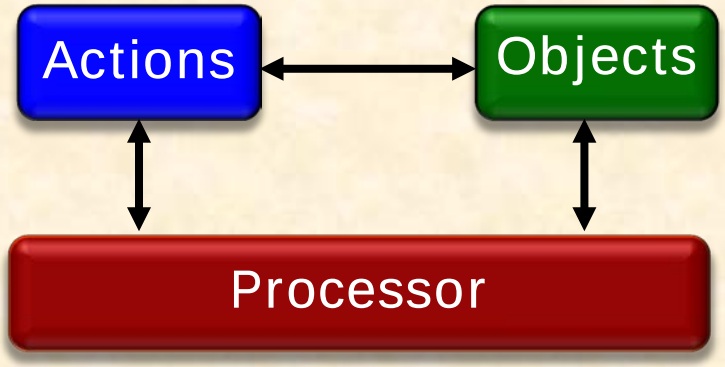
- (Static) type and module structure: class
- (Dynamic) data structure: object
- Inheritance for (static) reuse and (dynamic) binding

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



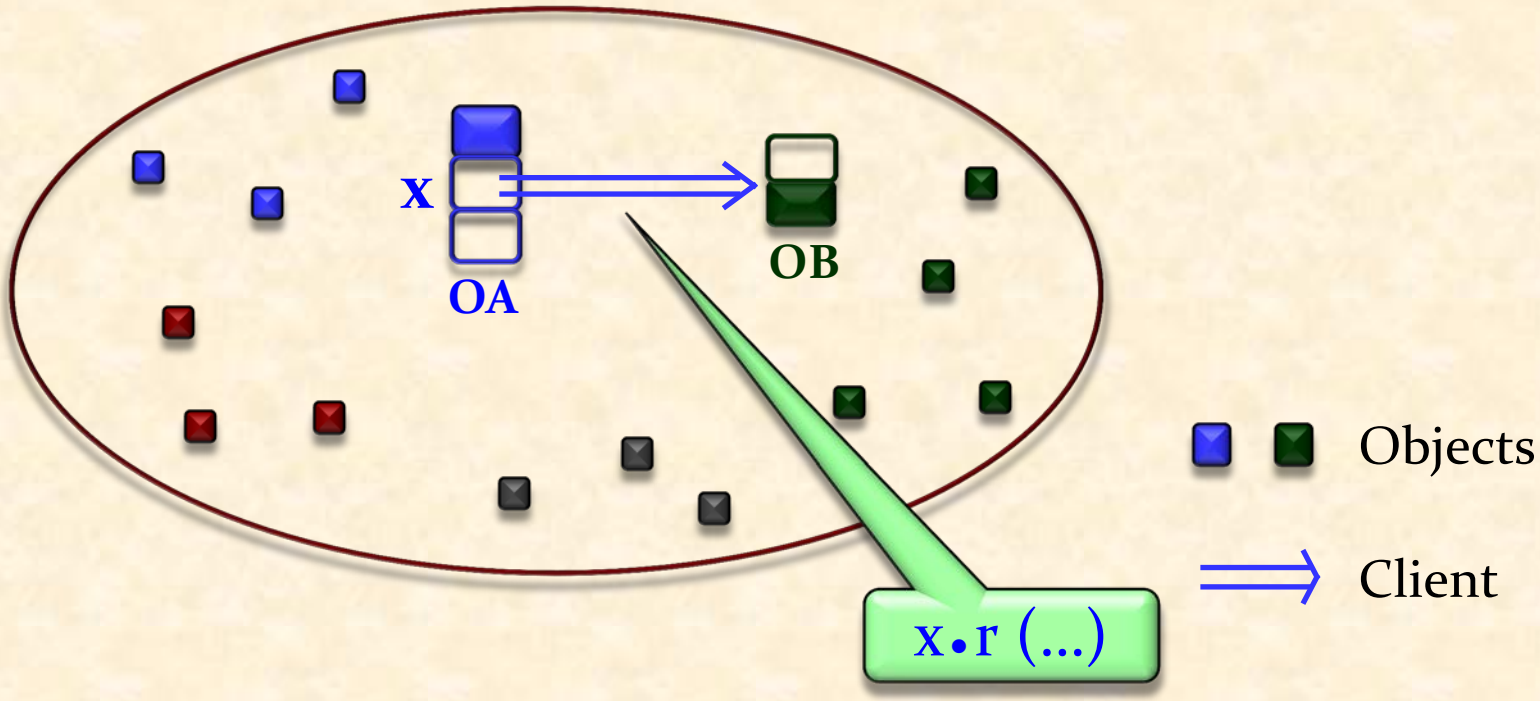
Will be mapped to computational resources

Choice 3: map object structure to processor structure

Fundamental operation in OO programming:

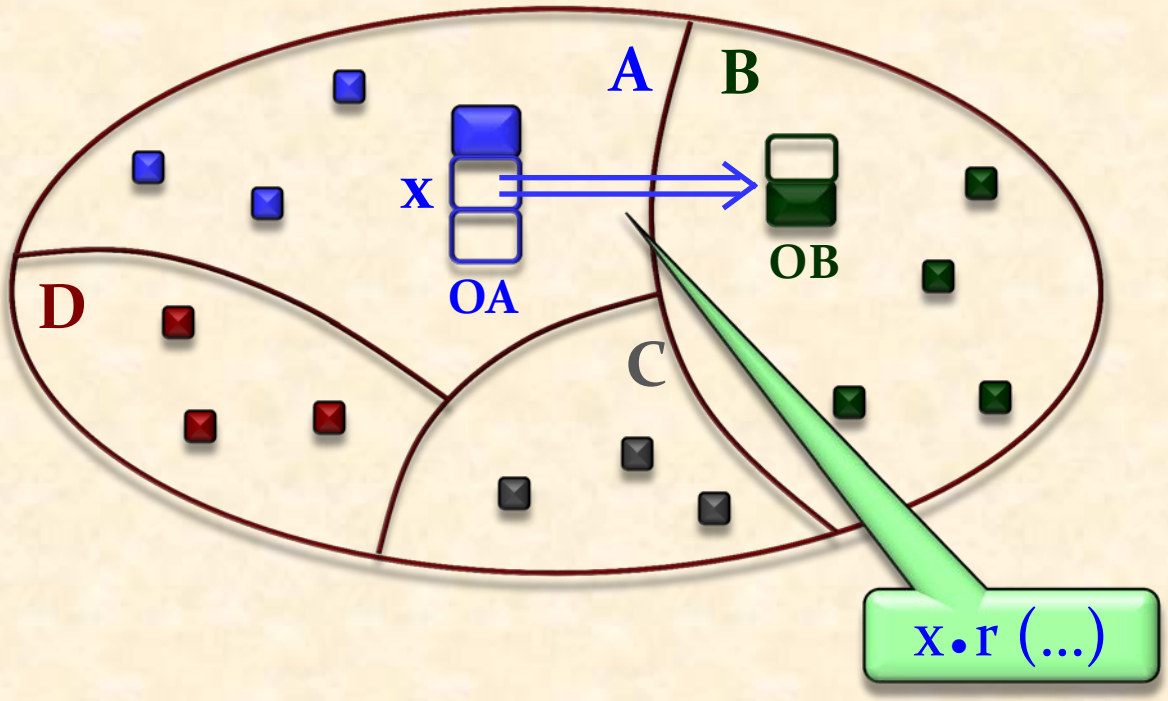
x.r (args)

Qualified call "targeted" to x
(also: "message passing"),



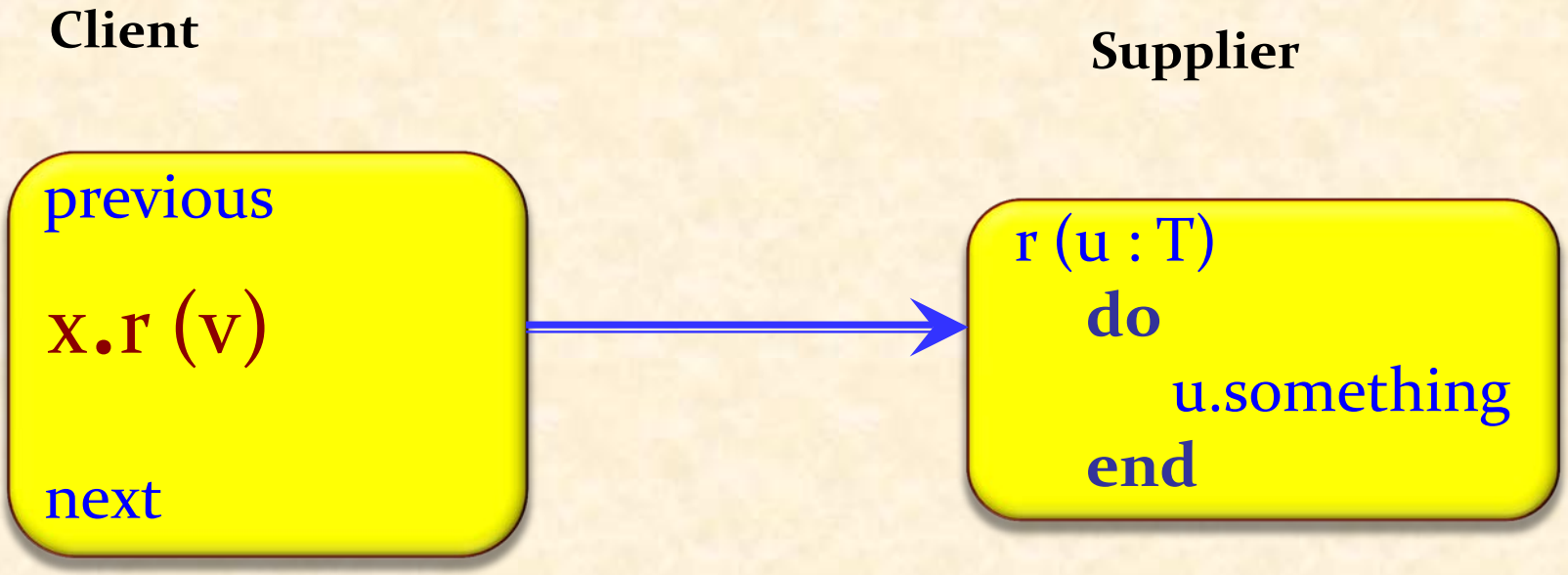
SCOOP handler rule

All calls targeted to a given object are performed by a single processor, called the object's handler



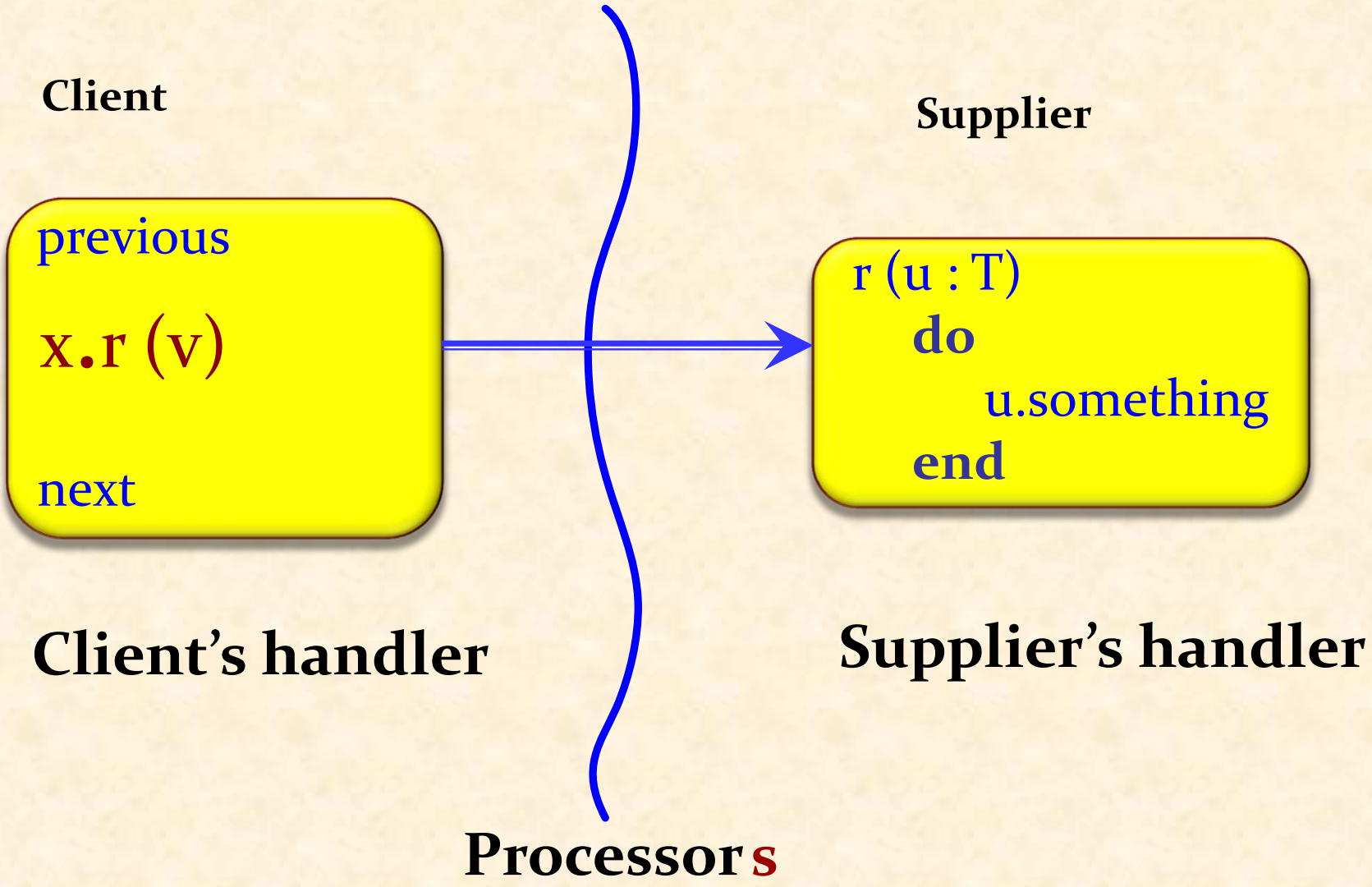
Choice 4: asynchronous command calls

Qualified call in the sequential world:



Processor

Qualified calls in a concurrent world



Call versus application

Benjamin Morandi

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

We must distinguish between:

xìr (args)

- 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

xìr (args)

No guarantee between > 2 processors

The two forms of O-O call

"Separate" means: possibly in a different region

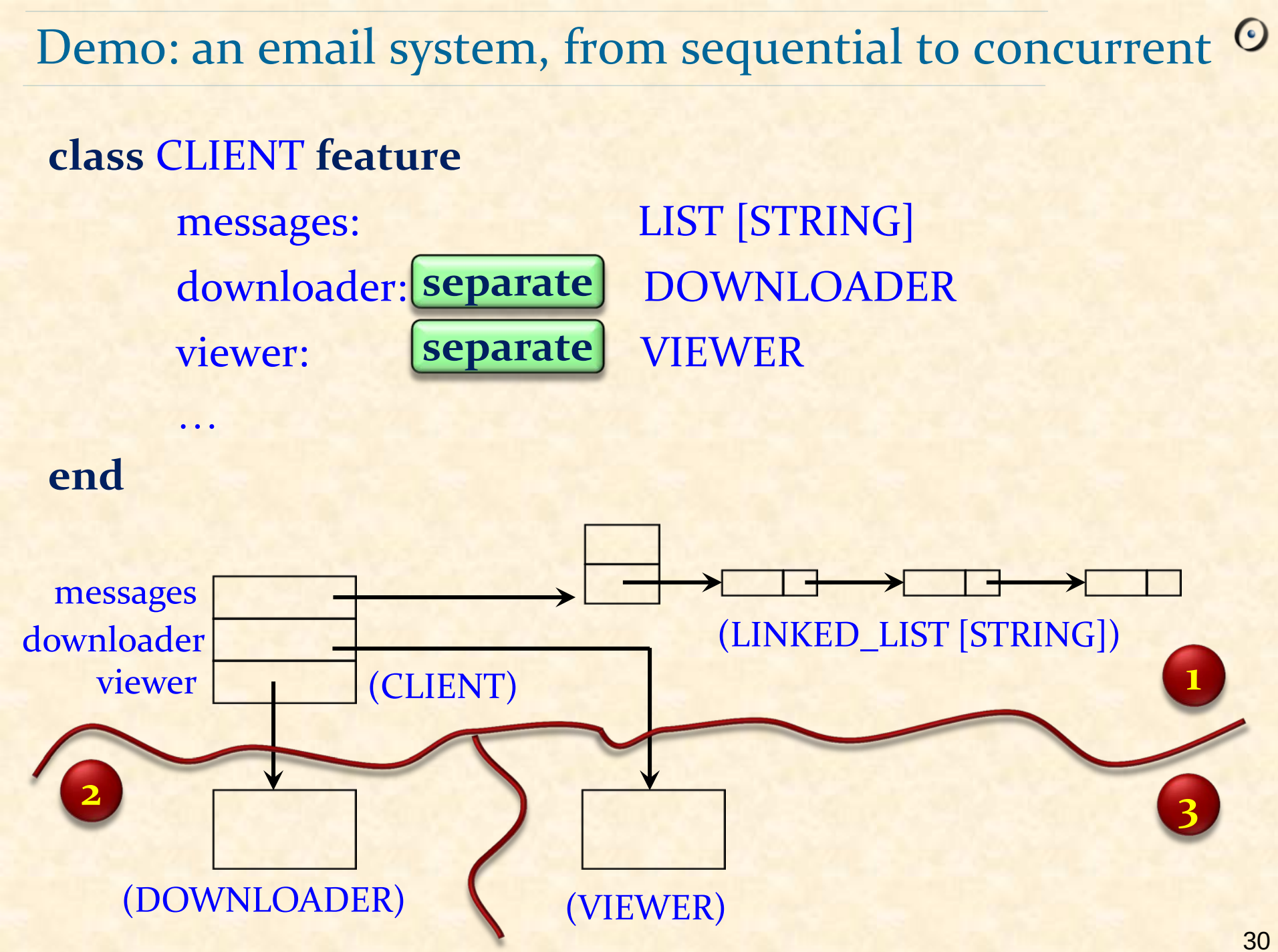
A command call *x.r* (*a*) is:

- 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

Demo: an email system, from sequential to concurrent



class CLIENT **feature**

messages: LIST [STRING]

downloader: **separate** DOWNLOADER

viewer: **separate** VIEWER

…

end

Choice 5: dynamic processor creation

With

x: **separate** T

the creation instruction

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

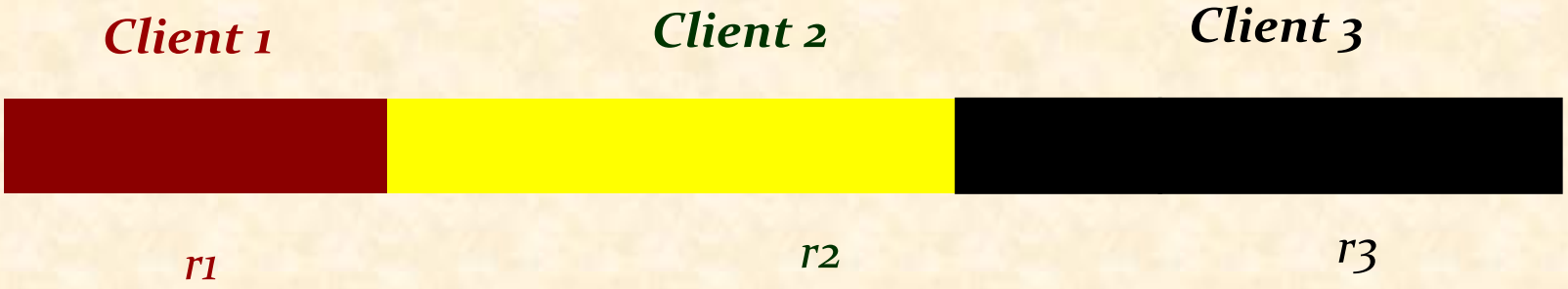
At any given time, at most one operation in progress on any given object

(In fact, on objects in any given region)

No intra-object concurrency

Reasoning about objects, concurrently

Only *n* proofs if *n* exported routines?



$$\frac{\{\text{INV and Pre}_r\}\quad \text{body}_r\quad \{\text{INV and Post}_r\}}{\{\text{Pre}_r'\}\quad \text{x.r (a)}\quad \{\text{Post}_r'\}}$$

Choice 7: multiple simultaneous object reservation

A call

xìr (a1, a2, ...)

will *wait* until it has been able to lock *all* the separate objects associated with the arguments *a1, a2, ...*

Guarantees mutual exclusion

Applies to locking *any number of objects*

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
```

Another example of mutual exclusion

This routine will lock *b*:

```
put (b : separate QUEUE [T]; value : T)
        -- Add value, FIFO-style, to b.
    do
        b.put (value)
    end
```

The buffer update *b.put (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

xìr (a1, a2, ...)

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?

```
x.command1 (u, v)

x.command2 (a, b)
x.command3
...
value := x.query1
```

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

```
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)

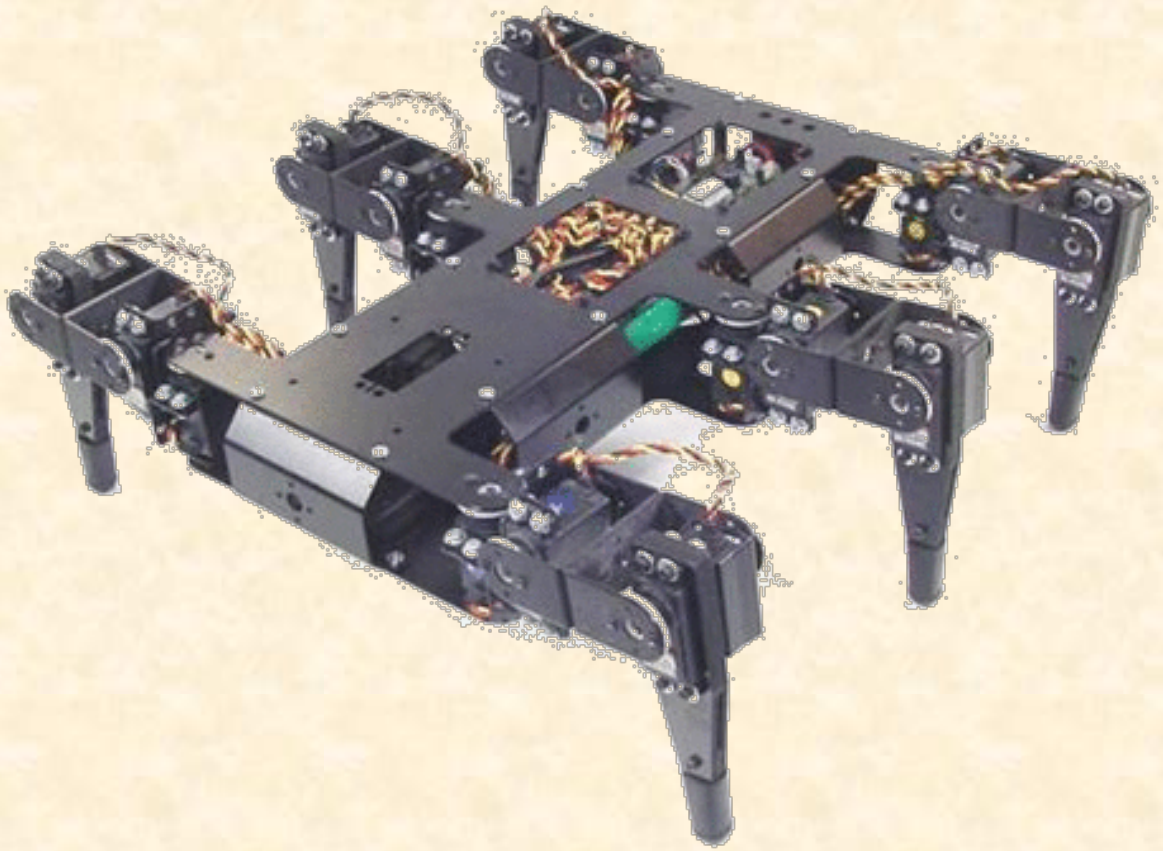
Precondition becomes **wait condition**

Example: bank transfer

```
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

Ganesh Ramanathan, Benjamin Morandi, IROS 2011



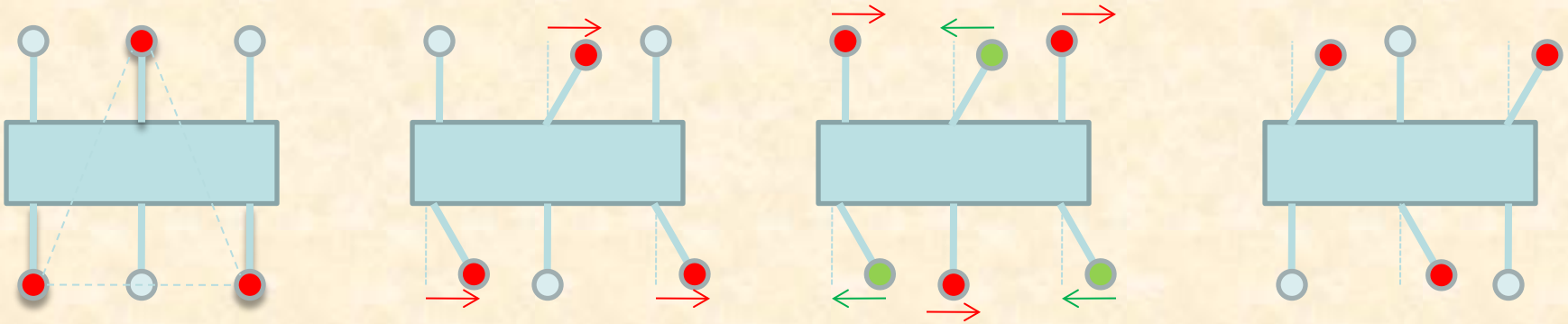
Roboscoop framework

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

SCOOP version

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.
        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();
                                            }
                                        }
```

Traitors and the SCOOP type system

Piotr Nienaltowski

a, b: PERSON

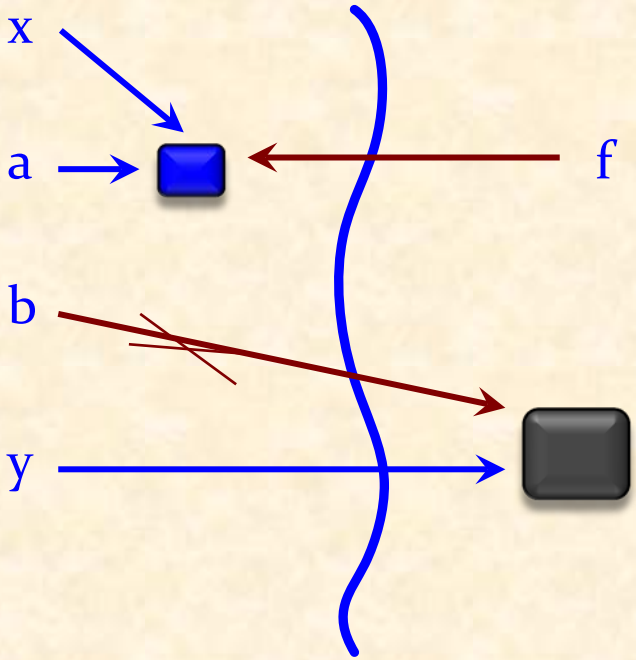
x, y: **separate** PERSON

x := a ✓

~~b := y~~

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

x!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…

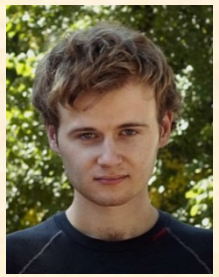










Open issues

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