Concurrent Programming Is Easy

Bertrand Meyer
ETH Zurich & Eiffel Software

Concurrent Programming School
Nijny Novgorod, October 2010

The issue

Concurrency everywhere:

- Multithreading
- Multitasking
- Networking, Web services, Internet
- Multicore

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

SCOOP mechanism

Simple Concurrent Object-Oriented Programming

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

Prototype implementation at ETH (since 2007); production implementation at Eiffel Software, released in steps starting November 2010

Most up-to-date descriptions:

- Piotr Nienaltowski's 2007 ETH PhD dissertation, see http://se.ethz.ch/people/nienaltowski/papers/thesis.pdf
- Benjamin Morandi, Sebastian S. Bauer and Bertrand Meyer *SCOOP - A contract-based concurrent object-oriented programming model*, in *Proc. of LASER summer school on Software Engineering 2007/2008*, ed. P. Müller, LNCS 6029, Springer-Verlag, July 2010, pages 41-90, see http://se.ethz.ch/~meyer/publications/concurrency/scoop_laser.pdf

Concurrent programming is supposed to be hard…

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()
```

What we write in sequential code

```
transfer (source, target: ACCOUNT;
          value: INTEGER)

        do
                source.withdraw (value)
                target.deposit (value)

        end
```

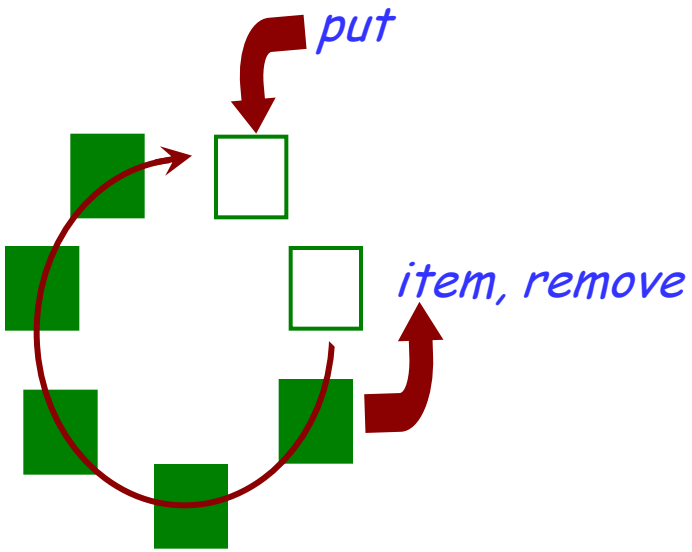
A better version

```
transfer (source, target:                ACCOUNT;
          value: INTEGER)
        require
                source.balance >= value
        do
                source.withdraw (value)
                target.deposit (value)
        ensure
                source.balance = old source.balance - value
                target.balance = old target.balance + value
        end
…
invariant
        balance >= 0
```

Make this concurrent!

```
transfer (source, target: separate ACCOUNT;
          value: INTEGER)
        require
                source.balance >= value
        do
                source.withdraw (value)
                target.deposit (value)
        ensure
                source.balance = old source.balance - value
                target.balance = old target.balance + value
        end

...
invariant
        balance >= 0
```




```
put (b: BUFFER [G]; v: G)
        -- Store v into b.
    require
        not b.is_full
    do
        ...
    ensure
        not b.is_empty
    end
```

```
my_queue: QUEUE [T]

...

if not my_queue.is_full then

        put (my_queue, t)

end
```

Previous advances in programming

	"Structured programming"	"Object technology"
Use higher-level abstractions	✓	✓
Helps avoid bugs	✓	✓
Transfers tasks to implementation	✓	✓
Lets you do stuff you couldn't before	NO	✓
Removes restrictions	NO	✓
Adds restrictions	✓	✓
Has well-understood math basis	✓	✓
Doesn't require understanding that basis	✓	✓
Permits less operational reasoning	✓	✓

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

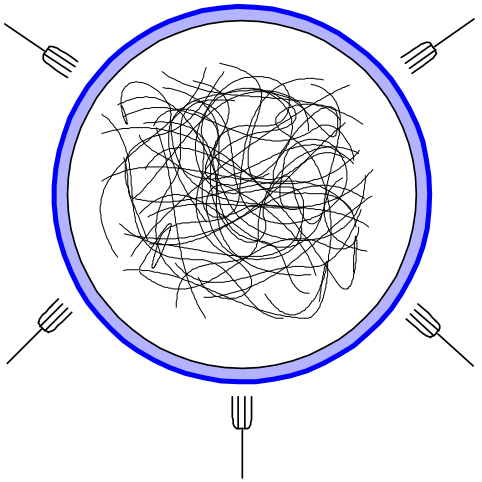
"*Objects are naturally concurrent*" (Milner)

- 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

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

```
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:

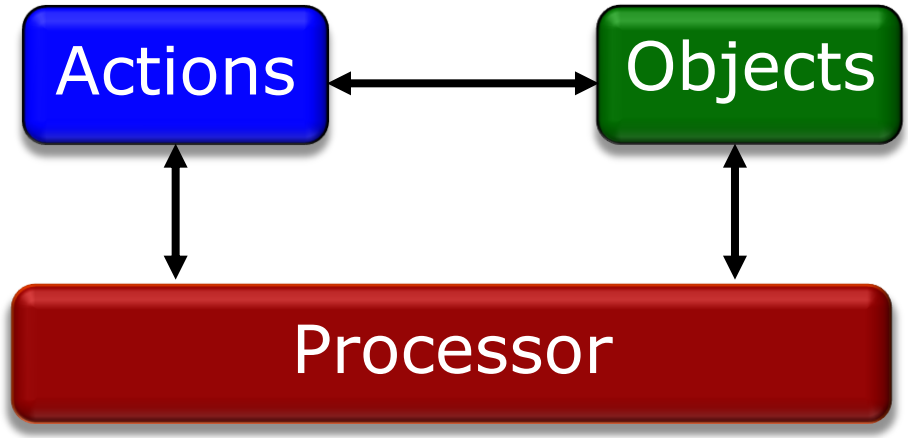
Listing 4.34: Tanenbaum's solution

```
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

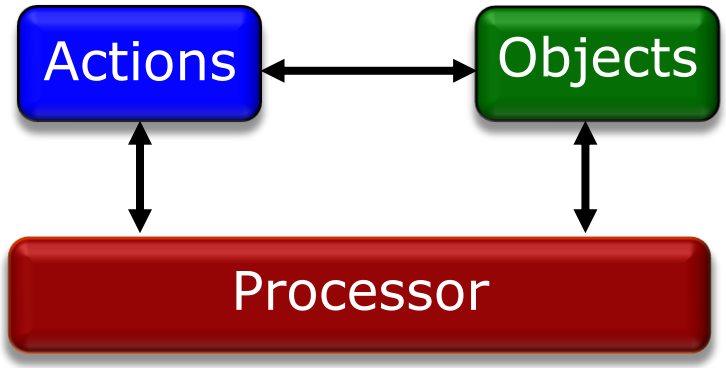


What makes an application concurrent?

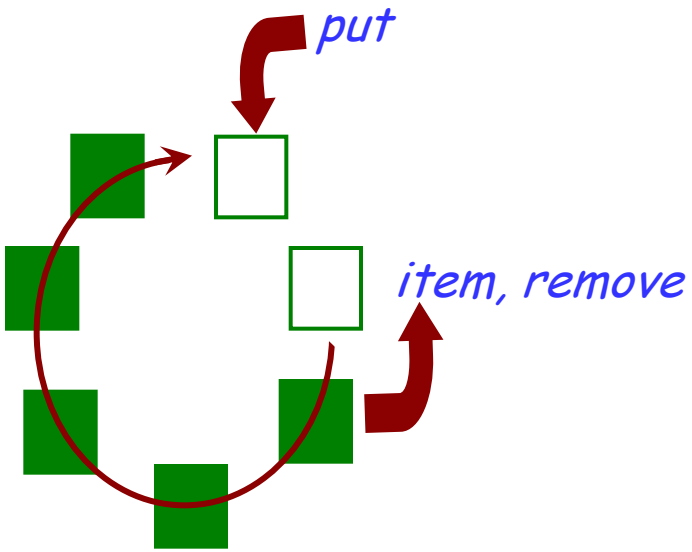
Processor:
Thread of control supporting sequential execution of instructions on one or more objects

Can be implemented as:

- Computer CPU
- Process
- Thread
- AppDomain (.NET) …



Will be mapped to computational resources




```
put (b: BUFFER [G]; v: G)
        -- Store v into b.
    require
        not b.is_full
    do
        ...
    ensure
        not b.is_empty
    end
```

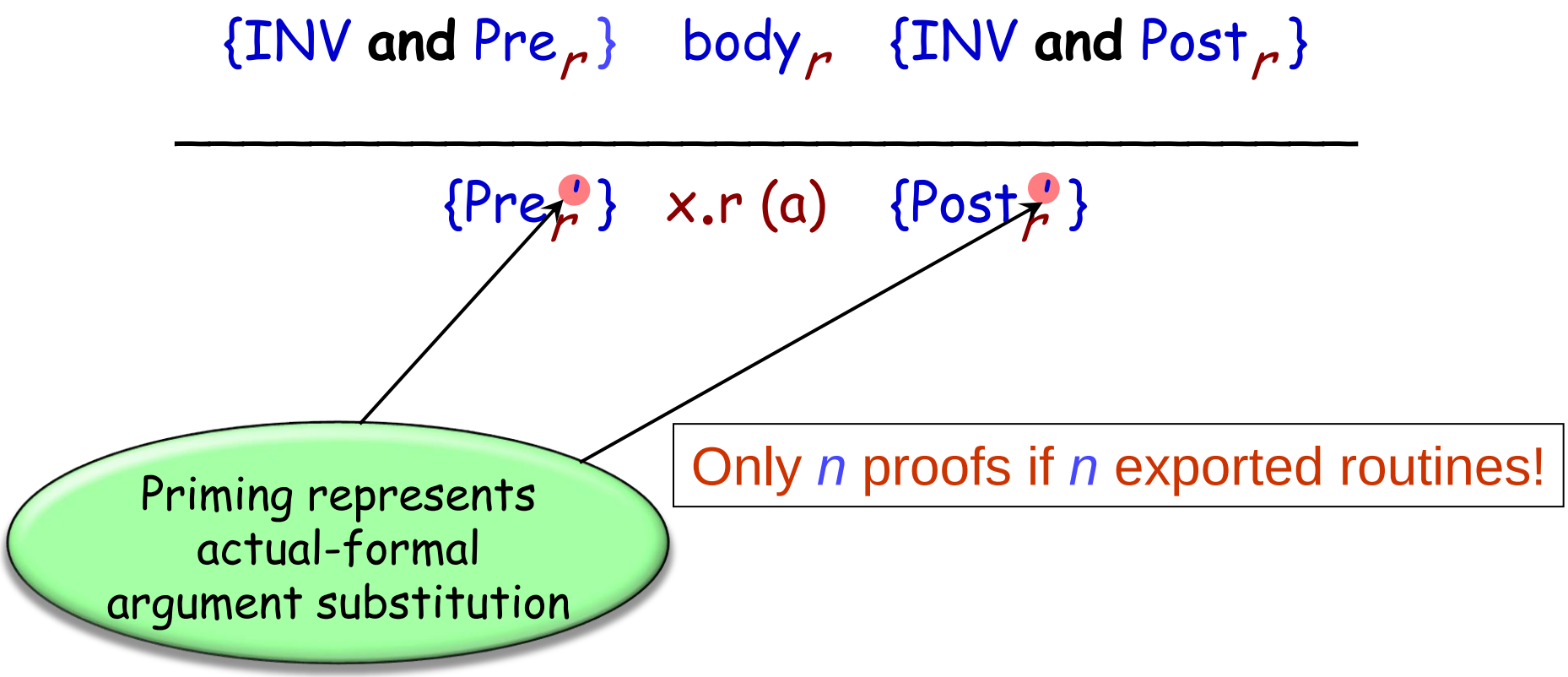
```
my_queue: BUFFER [T]

...

if not my_queue.is_full then
```

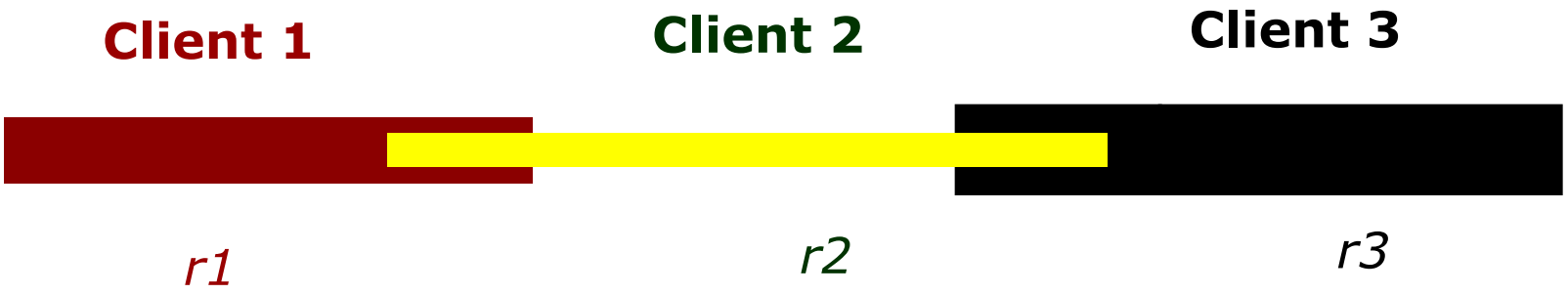
```
        put (my_queue, t)
end
```

Reasoning about objects: sequential



In a concurrent context

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



No overlapping!

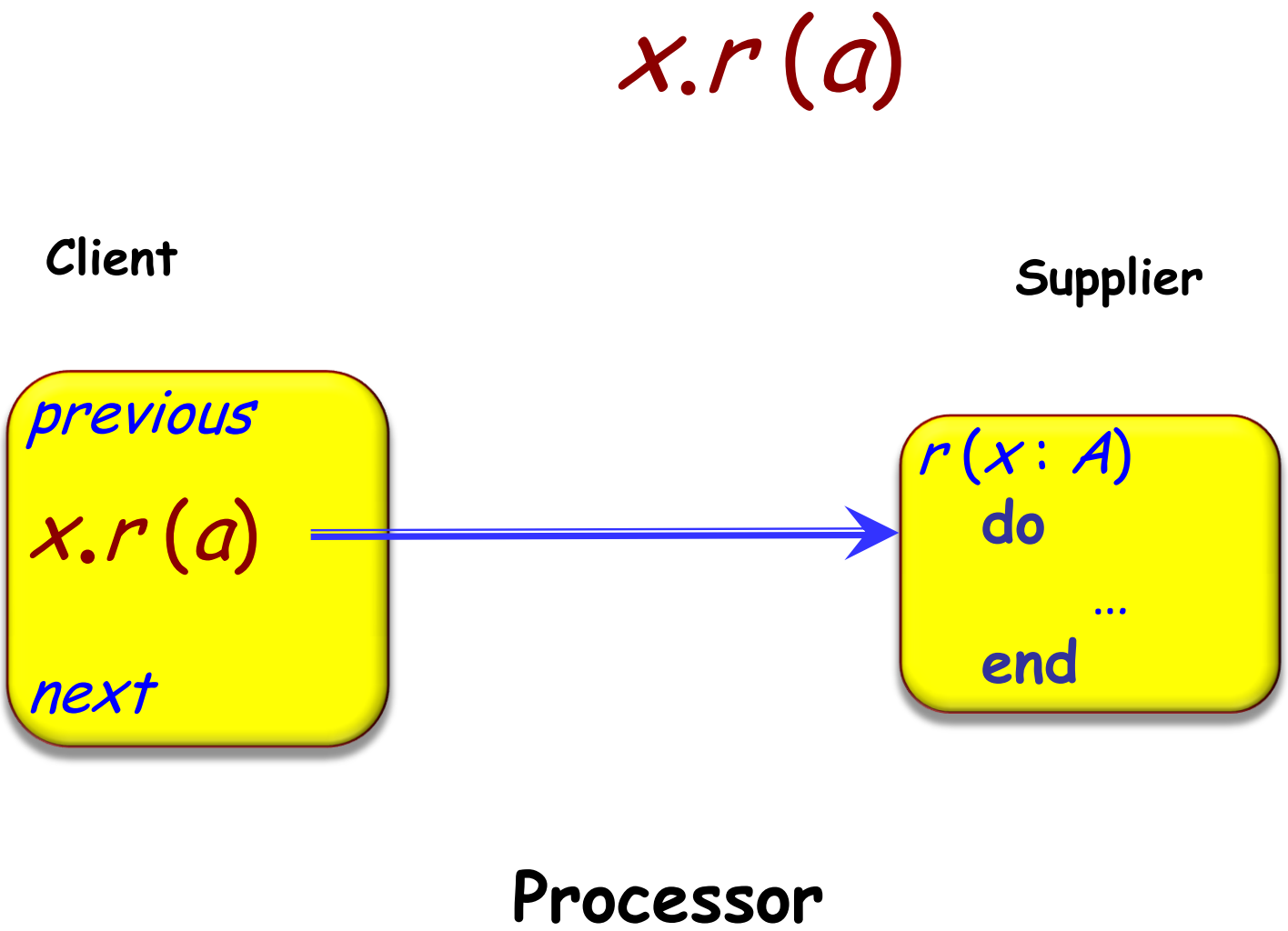
$$\frac{\{\text{INV and } Pre_r\} \quad body_r \quad \{\text{INV and } Post_r\}}{\{Pre_{r'}\} \quad x.r\ (a) \quad \{Post_{r'}\}}$$

SCOOP rules

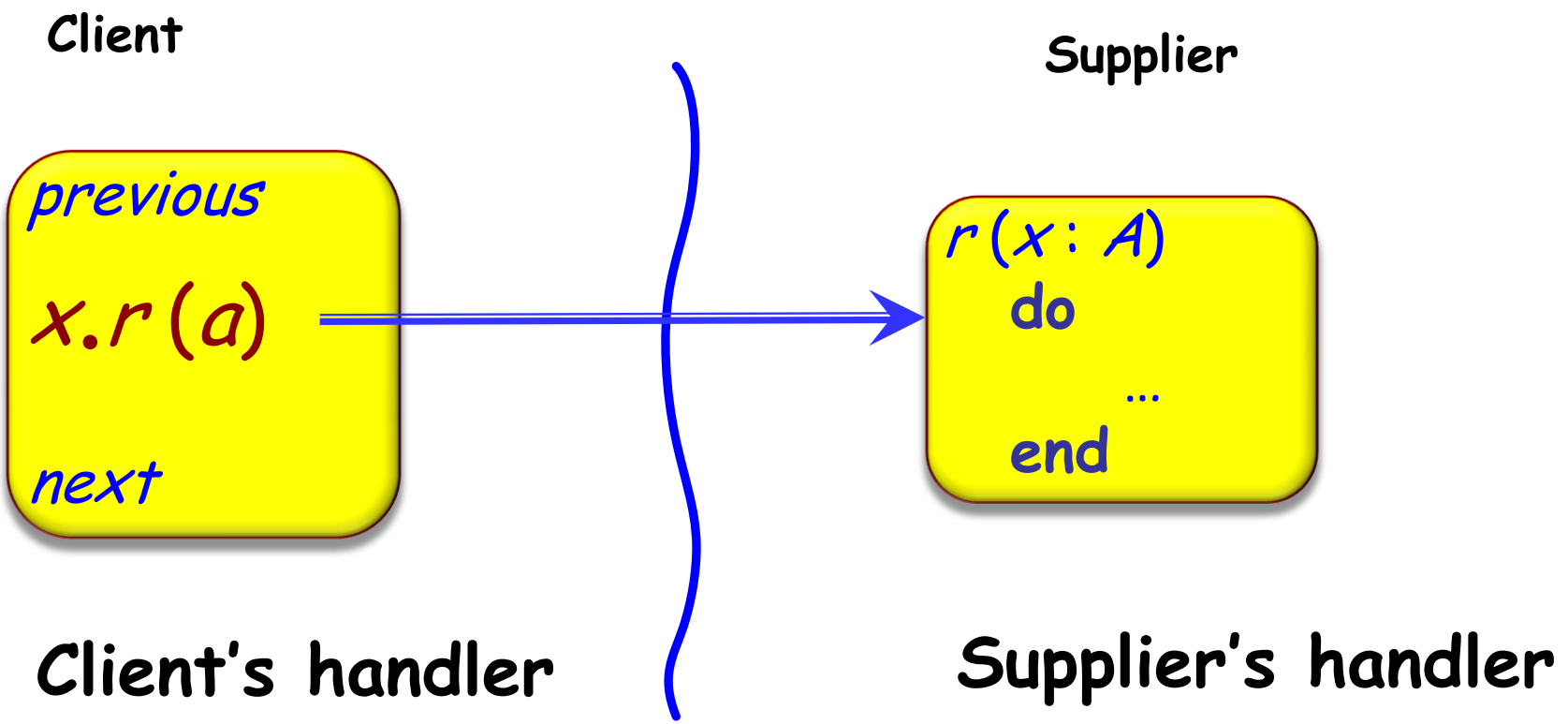
- One processor per object: "handler"

- At most one feature (operation) active on an object at any time

Feature call: sequential



Feature call: asynchronous



The fundamental difference

To wait or not to wait:

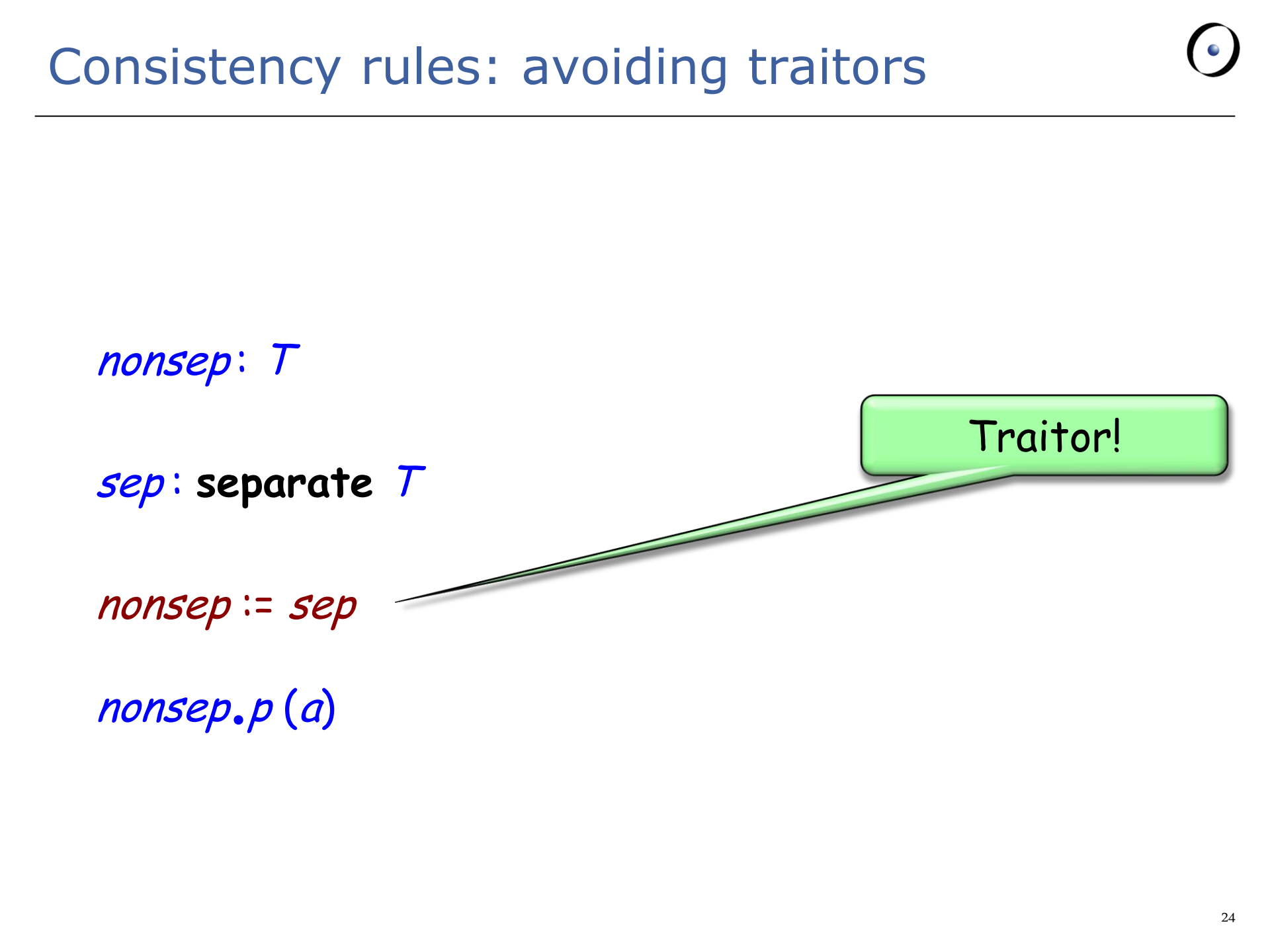
- If same processor, synchronous
- If different processor, asynchronous

Difference must be captured by syntax:

- x: T
- x: **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



Trusting what you read

my_buffer: **separate** *QUEUE*[*T*]

…

my_buffer.put (*a*)

… Instructions not affecting the buffer…

y := *my_buffer.item* ⟵ ?

Access control policy

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

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

Access control policy

Target of a separate call must be formal argument of enclosing routine:

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

To use separate object:

```
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* (...) where *x* is separate

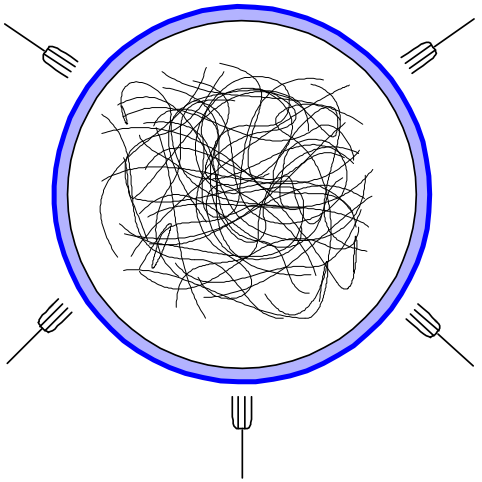
Wait rule

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:

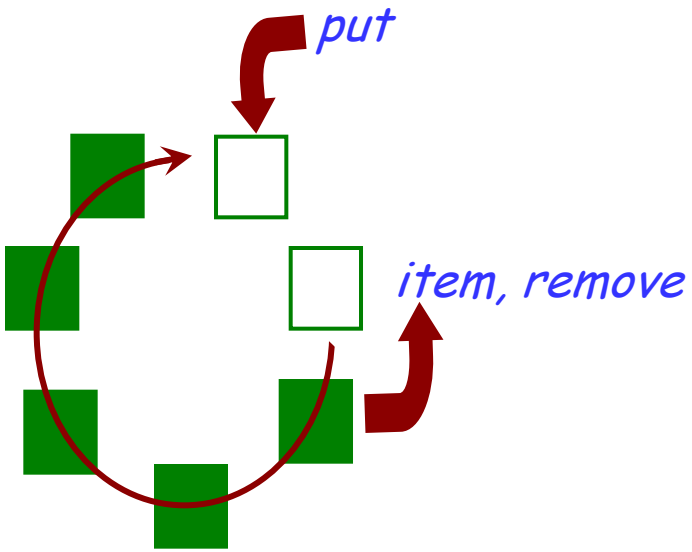
```
x.f
x.g (a)
y.f
...
value := x.some_query
```

Lazy wait (Denis Caromel, wait by necessity)

Contracts

```
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 )
```




```
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
```

Contracts

```
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 )
```

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?

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

Wait condition

Correctness condition

```
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?

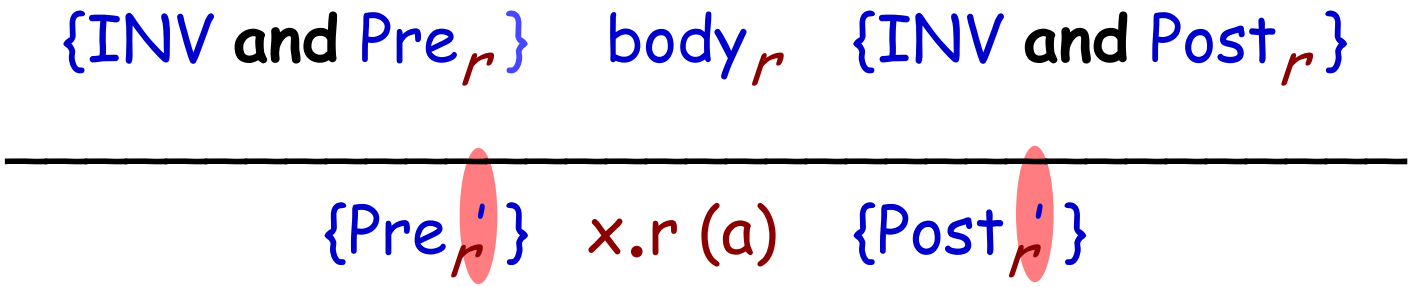
```
zurich, novgorod : separate LOCATION

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, novgorod)
do_local_stuff
get_result (zurich)
```

Should we wait for *zurich.is_ready*?

Reasoning about objects: sequential



Only *n* proofs if *n* exported routines!

Refined proof rule (partial correctness)

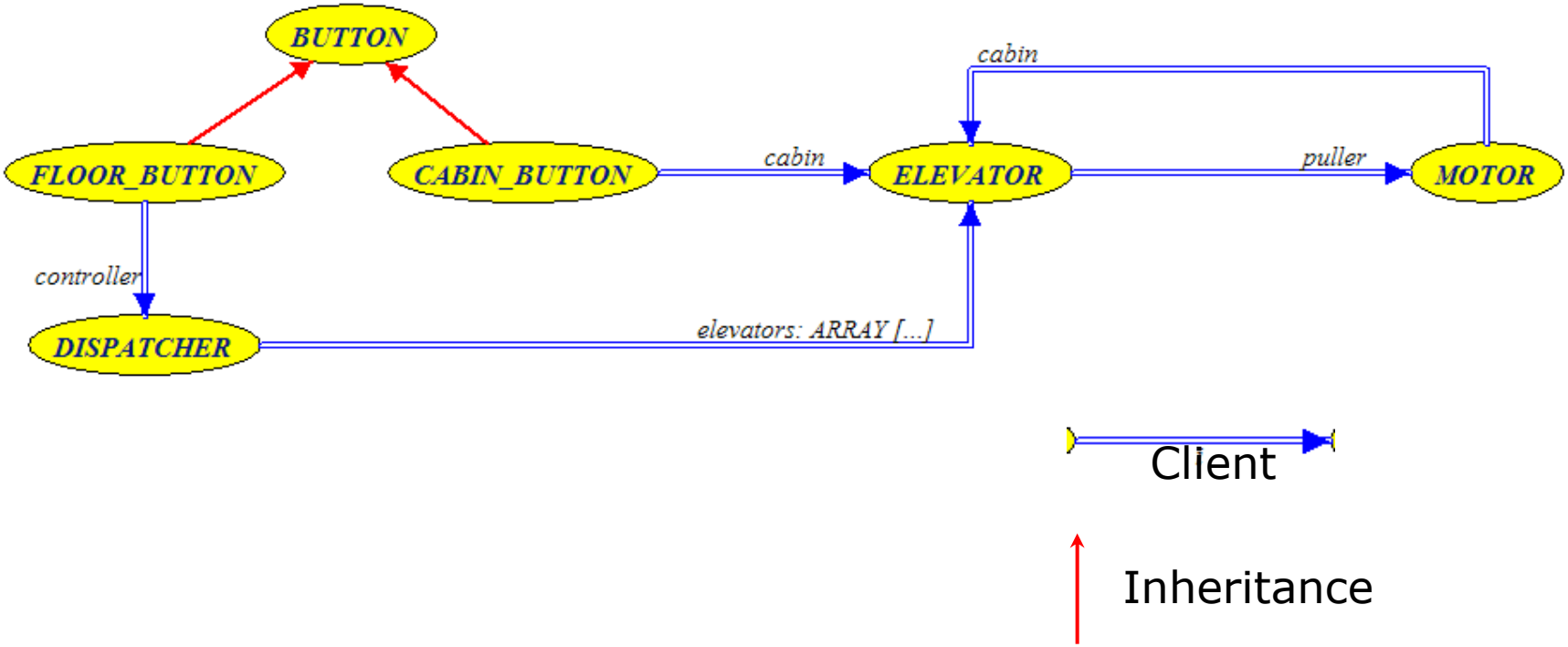
$$\frac{\{INV \wedge Pre_r(x)\}\ body_r\ \{INV \wedge Post_r(x)\}}{\{Pre_r(a^{cont})\}\ \ e.r(a)\ \ \{Post_r(a^{cont})\}}$$

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

Other aspects

What if a separate call, e.g. in

```
r (a : separate T)
        do
                a.f
                a.g
                a.h
        end
```

causes an exception?

Status

- All of SCOOP except exceptions and duels implemented
- Implementation 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
- Concurrency should be easy: programmers need to sleep better!