ETH Zürich

Chair of Software Engineering

Concepts of Concurrent Computation

Bertrand Meyer
Sebastian Nanz

Lecture 6: an overview of SCOOP

The issue that SCOOP addresses

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

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

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	✓	✓

The chasm

Theoretical models, process calculi (see forthcoming lectures)

Elegant theoretical basis, but

- Remote from the ordinary practice of programming
- Handle concurrency aspects only

Practice of concurrent & multithreaded programming

- Low-level, e.g. threads, semaphores
- Poorly connected with rest of programming model (O-O structure of modern programs)

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 (2005-2008)

Recent production implementation at Eiffel Software, part of EiffelStudio

Recent descriptions: Piotr Nienaltowski's 2007 ETH PhD dissertation; Morandi, Nanz, Meyer (2011)

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 programs

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

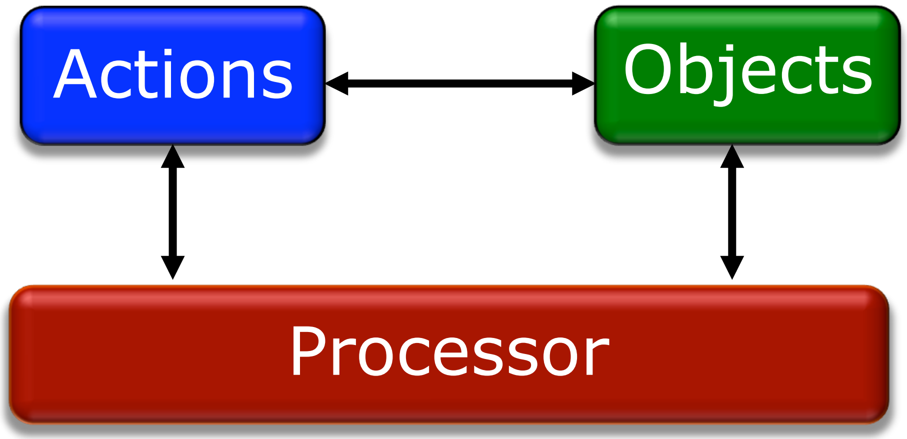
SCOOP narrows down the distinction between sequential and concurrent programming to five key properties, studied next:

- **(A)** Single vs multiple "processors"
- **(B)** Synchronous vs asynchronous calls
- **(C)** Semantics of argument passing
- **(D)** Semantics of resynchronization (lazy wait)
- **(E)** 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

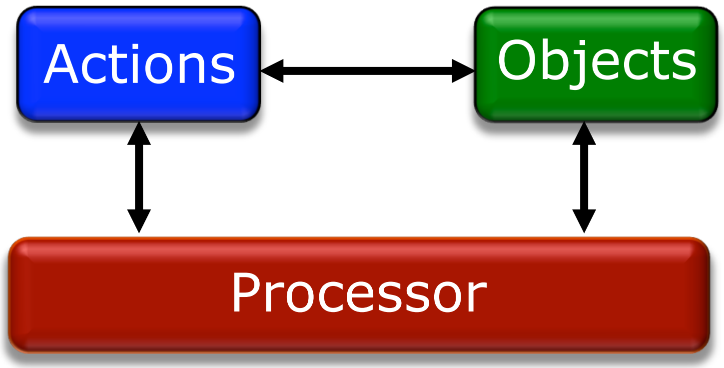
Concurrent: any number of processors

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

Reasoning about objects: sequential

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

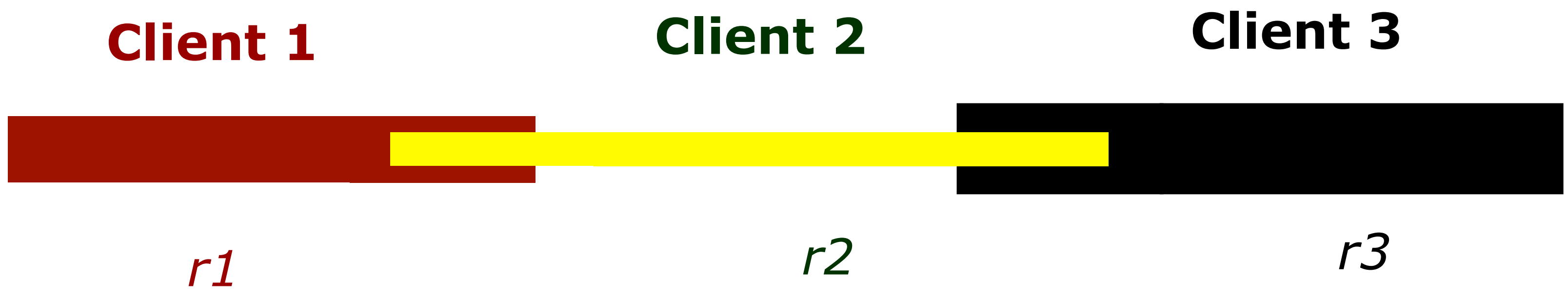
$$\frac{\{INV \textbf{ and } Pre_r\} \quad body_r \quad \{INV \textbf{ and } Post_r\}}{\{Pre_r'\} \quad x.r\ (a) \quad \{Post_r'\}}$$

Priming represents actual-formal argument substitution

The concurrent version of this rule will come later!

In a concurrent context

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



No overlapping!

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

SCOOP restriction: one handler per object

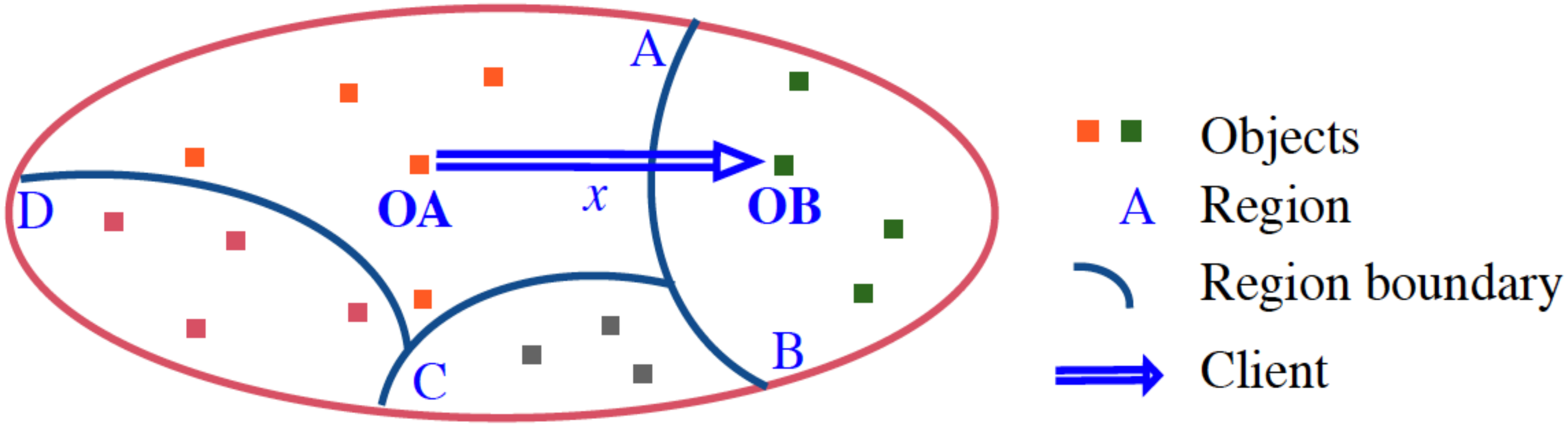
- One processor per object: "handler"

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

Regions

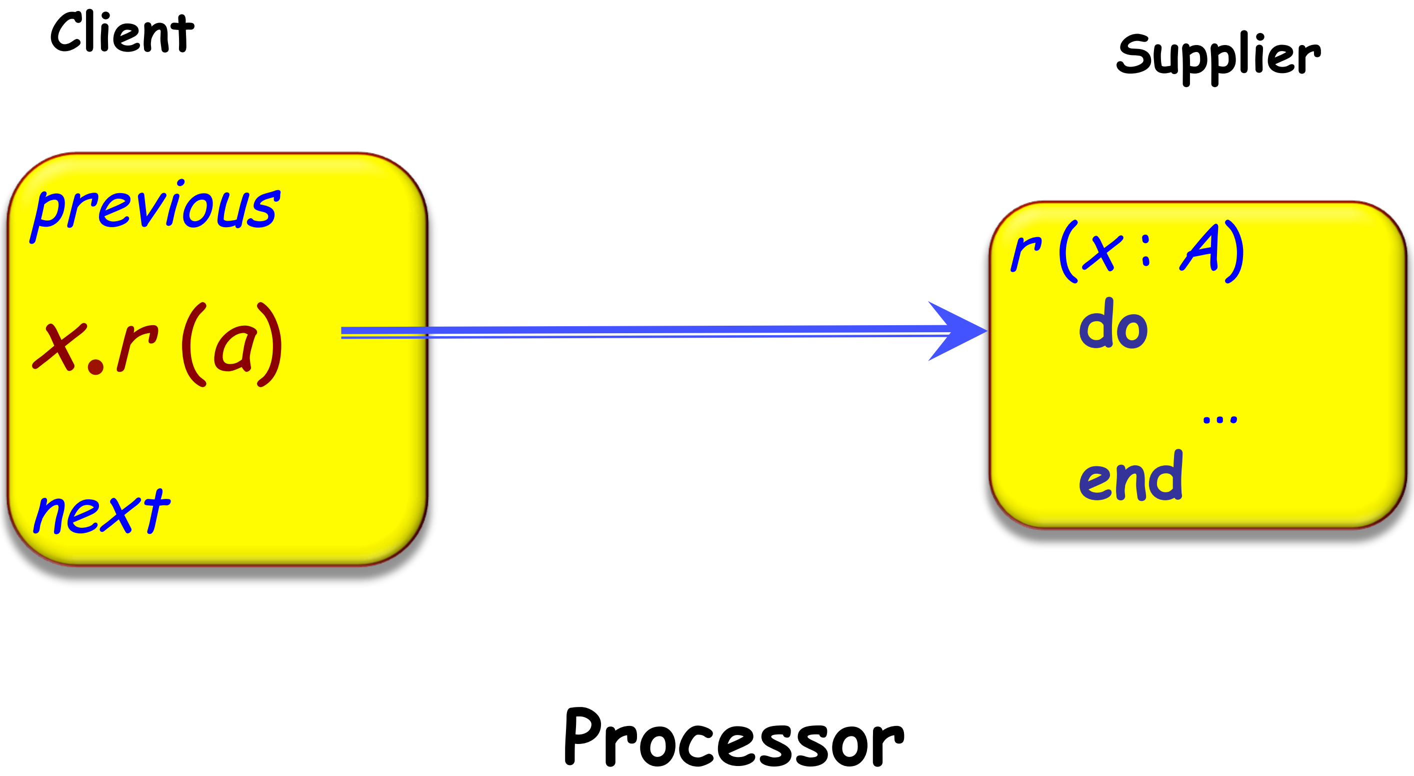
The notion of handler implies a partitioning of the set of objects:

- The set of objects handled by a given processor is called a *region*
- Handler rule implies one-to-one correspondence between processors and regions.

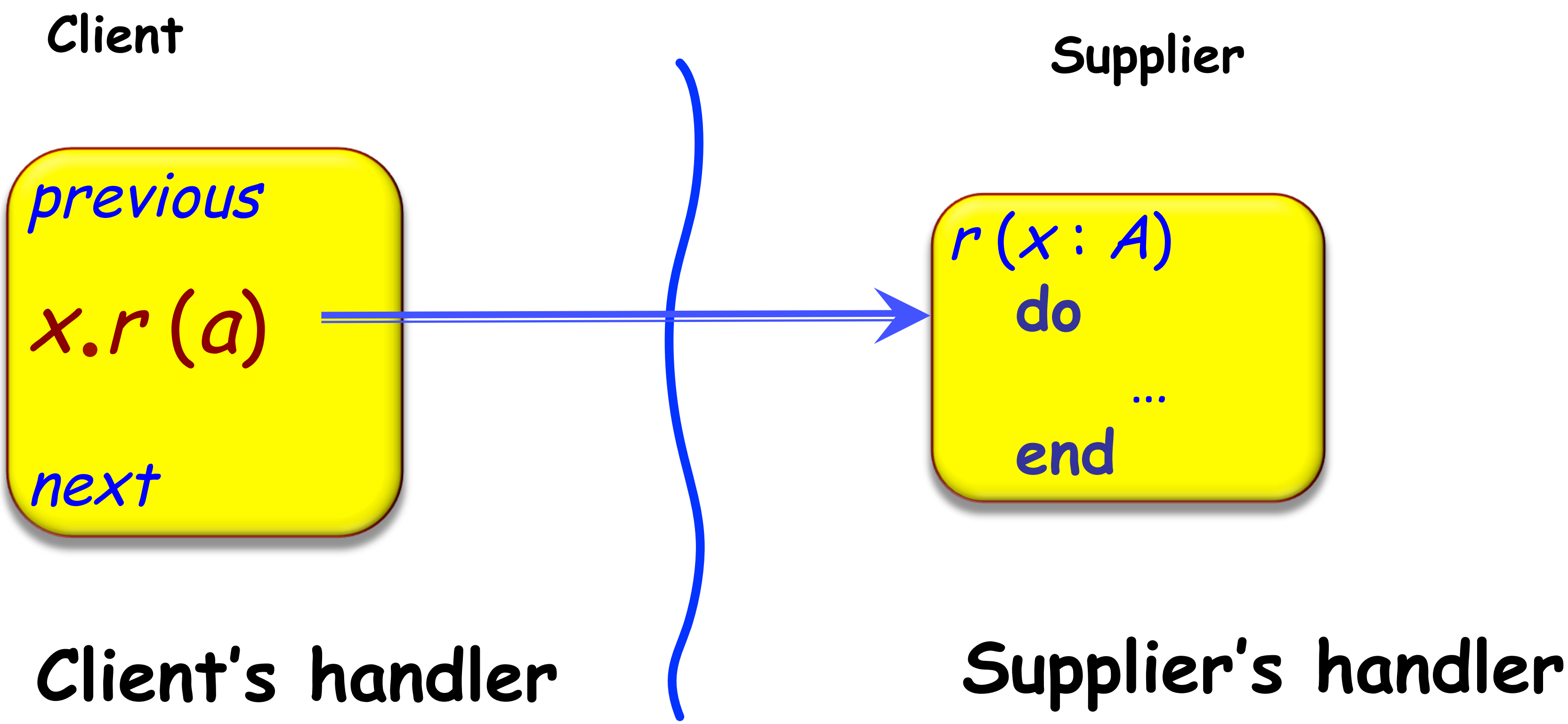


(B) The sequential view: O-O feature calls

$x.r\,(a)$



(B) The concurrent form of call: asynchronous



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:

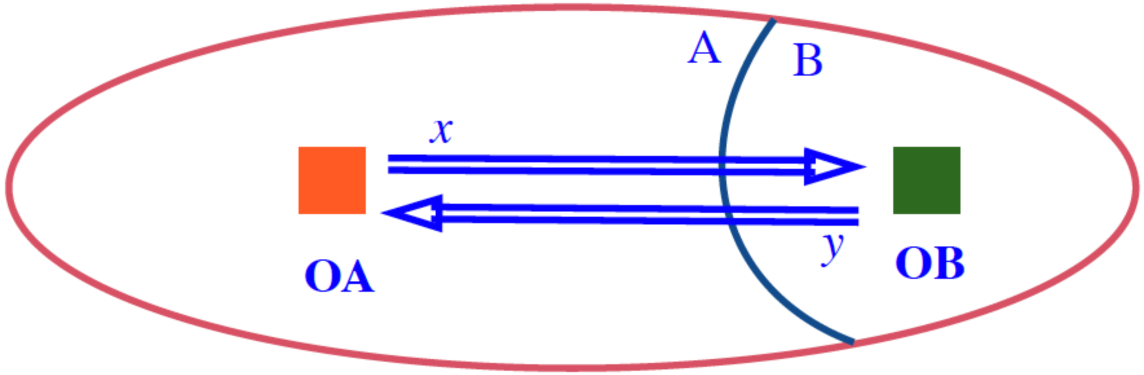
- x: T
- x: **separate** T -- Potentially different processor

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 does not specify processor: only states that the object *might* be handled by a different processor



- In class *A*: *x*: **separate** *B*
- In class *B*: *y*: **separate** *A*

In some execution the value of x.y might be a reference to an object in the current region (including **Current** itself)

Call vs application

With asynchrony we must distinguish between feature call and feature application

The execution

x . 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

nonsep : *T*

sep : **separate** *T*

nonsep := *sep*

nonsep.*p* (*a*)

Traitor!

More traitor protection through the type system! (next lectures)

Trusting what you read ("reasonability")

```
remote_stack: separate STACK [T]

...

remote_stack.put (a)

... Instructions not affecting the buffer...

y := remote_stack.item        ←————————  ?
```

(C) Access control policy

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

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

(C) Access control policy

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

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

To use separate object:

```
my_buffer: separate QUEUE [INTEGER]
create my_buffer
put (my_buffer , 10)
```

(C) 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

(C) Wait rule

A routine call guarantees exclusive access to the handlers (the processors) of all separate arguments

a_routine (nonsep_a, nonsep_b, sep_c, sep_d, sep_e)

Exclusive access to *sep_c, sep_d, sep_e* within *a_routine*

Background for this rule: "reasonability" again

An example: from sequential to concurrent

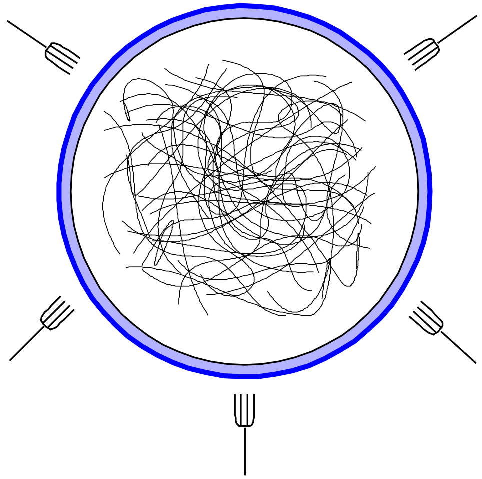
```
transfer (source, target: separate ACCOUNT;
            amount: INTEGER)
        -- Transfer amount from source to target.
    require
         source.balance >= amount
    do
        source.withdraw  (amount)
        target.deposit   (amount)
    ensure
        source.balance = old source.balance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

Dining philosophers in SCOOP

```
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

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

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

def put_fork(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()
```

A *PROCESS* library class

SCOOP integrates inheritance and other O-O techniques with concurrency, seamlessly and without conflicts ("inheritance anomaly")

No need for built-in notion of **active object**: it is **programmed** through a library class such as *PROCESS* :

```
class process feature
    setup do end
    step do end
    over : BOOLEAN
    tear_down do end
    live
            do
                    from setup until over loop step end
                    tear_down
            end
    end
end
```

(C) 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

- Why? See computational model ...

Semantics vs implementation

Older SCOOP literature (OOSC, Nienaltowski, Morandi...) 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 current implementation performs these optimizations

```
f (a, b, c: separate T)
    do
        something_else
        a.r
        b.s
    end
```

No need to wait for a and b until here

No need to wait for c!

(D) Resynchronization: lazy wait

How do we resynchronize after asynchronous (separate) call?

No explicit mechanism!

The client will wait only when it needs to:

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

Wait here!

Lazy wait (also known as wait by necessity)

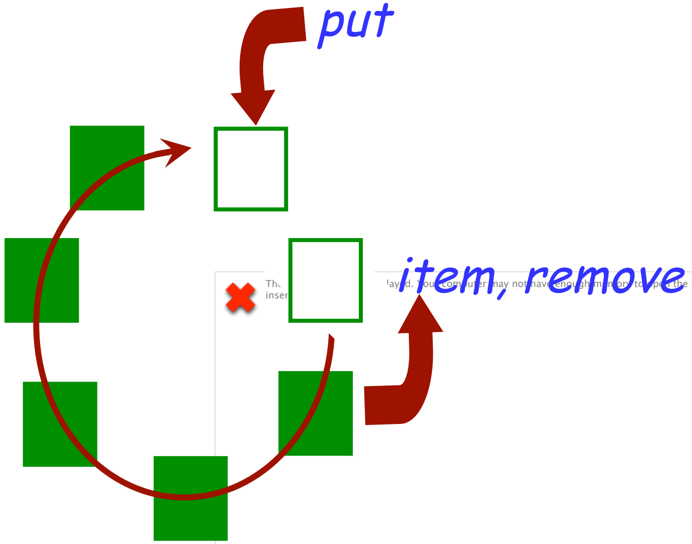
(D) Synchrony vs asynchrony revisited

For a separate target *x*:

- *x.command* (...) is asynchronous
- *v* := *x.query* (...) is synchronous

(E) Contracts

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




```
my_queue : BUFFER [T]
...

if not my_queue.is_full then
```

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

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

(E) 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 )
```

(E) 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**

(E) Full synchronization rule

A call with separate arguments waits until:

- The corresponding objects are all available
- Preconditions hold

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

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

What about postconditions?

zurich, lausanne : **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, lausanne)
do_local_stuff
get_result (zurich)
```

Should we wait for *zurich.is_ready*?

Reasoning about objects: sequential

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

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

SCOOP highlights

- Close connection to O-O modeling
- Natural use of O-O mechanisms such as inheritance
- Built-in guarantee of no data races
- 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