

Chair of Software Engineering



Concurrent Programming Is Easy

Bertrand Meyer ETH Zurich & Eiffel Software

Concurrent Programming School Nijny Novgorod, October 2010

The issue

 \bigcirc

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?

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 <u>http://se.ethz.ch/people/nienaltowski/papers/thesis.pdf</u>
- 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

```
def get_fork(i):
 2
       mutex.wait()
 3
       state[i] - 'hungry'
 4
       test(i)
 5
       mutex.signal()
       sem[i].wait()
 6
 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
   def test(i):
15
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

 ACCOUNT;

do

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





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



```
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



my_queue: BUHFIER[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	\checkmark	\checkmark
Helps avoid bugs	\checkmark	\checkmark
Transfers tasks to implementation	\checkmark	\checkmark
Lets you do stuff you couldn't before	NO	\checkmark
Removes restrictions	NO	\checkmark
Adds restrictions	\checkmark	\checkmark
Has well-understood math basis	\checkmark	\checkmark
Doesn't require understanding that ba	sis 🗸	\checkmark
Permits less operational reasoning	\checkmark	\checkmark

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

 \bigcirc

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

"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





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

```
def get_fork(i):
 1
 2
       mutex.wait()
 3
       state[i] - 'hungry'
 4
       test(i)
 5
       mutex.signal()
       sem[i].wait()
 6
 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
   def test(i):
15
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()
```

To perform a computation is

- > To apply certain actions
- > To certain objects
- > Using certain processors



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



my_queue: BUFFER[T]

if not my_queue.is_full then



put(my_queue, t)

end

...





 \bigcirc

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





> One processor per object: "handler"

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

Feature call: sequential



Processor



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 -- <u>Potentially</u> different processor

Fundamental semantic rule: x.r(a) waits for nonseparate x, doesn't wait for separate x.



nonsep.p(a)

my_buffer: separate QUEUE[T]

my_buffer.put(a)

...

... Instructions not affecting the buffer...

y := my_buffer.item ←



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

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)

The target of a separate call must be an argument of the enclosing routine

Separate call: x.f(...) where x is separate

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





 \bigcirc

No explicit mechanism needed for client to resynchronize with supplier after separate call.

The client will wait only when it needs to:

Lazy wait (Denis Caromel, wait by necessity)

```
put (buf : BUFFER [INTEGER]; v : INTEGER)
           -- Store vinto buffer.
     require
           not buf.is_full
           v > 0
     do
           buf.put(v)
     ensure
           not buf.is_empty
     end
```

... put(my_buffer, 10)

my_queue: BUFFER[T]

if not my_queue.is_full then

put(my_queue, t)

end

...

... put(my_buffer, 10) 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?

- > 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

{INV and
$$Pre_{r}$$
} body_r {INV and $Post_{r}$ }
{Pre_r} x.r (a) {Post_r}

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

{ $INV \land Pre_r(x)$ } body_{r}{ $INV \land Post_r(x)$ }

 $\{Pre_r(a^{cont})\} e.r(a) \{Post_r(a^{cont})\}$

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

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

Implementation: integrating into EiffelStudio Performance evaluation

Theory:

- Deadlock prevention and detection
- Less restrictive model (see STM)
- Transactions
- Full-fledged semantics
- Distributed SCOOP, Web Services

- 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!