Concurrent Programming Is Easy

Bertrand Meyer
ETH Zurich & Eiffel Software

Future Of Software Engineering
ETH Zurich Symposium, November 2010
Two topics

1. Concurrent Programming Is Easy

2. Developments at the Chair of Software Engineering: an overview
Moore’s Law

- Transistor count still rising
- Clock speed flattening sharply

Source: Intel
Concurrency

Multicore, Internet, high-performance computing...

Multi-core processing is taking the industry on a fast-moving and exciting ride into profoundly new territory. The defining paradigm in computing performance has shifted inexorably from raw clock speed to parallel operations and energy efficiency. (Intel)

Faster Chips Are Leaving Programmers in Their Dust (John Markoff, NY Times, 2007)

Multicore processors represent one of the largest technology transitions in the computing industry today, with deep implications for how we develop software. (Rick Rashid)

Multicore: This is the one which will have the biggest impact on us. We have never had a problem to solve like this. A breakthrough is needed. (Bill Gates)
Concurrent programming is supposed to be hard...

Listing 4.33: Variables for Tanenbaum’s solution

```python
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

```python
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()
```
The chasm

Qsort = last → end → Qsort
   □ in?x → ((INx // up:Qsort) //down:Qsort)

Inx = in?y → ((up.in!y → INx) {y > x} (down.in!y → INx))
   □ last → up.last → down.last → OUTAx

OUTAx = up.out?y → out!y → OUTAx
   □ up.end → out!x → OUTB

OUTB = down.out?y → out!y → OUTB
   □ down.end → end → X

X = last → end → X
   □ in?x → INx

CSP, Source: A.W. Roscoe
The chasm

Listing 4.33: Variables for Tanenbaum’s solution

```python
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

```python
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(right(i)) != 'eating' and
18        state(right(i)) != 'eating':
19            state[i] = 'eating'
20            sem[i].signal()
```
Wrong 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
Active objects can be programmed

```process
deferred class PROCESS feature
    live
do
    from setup until over loop step end
    tear_down
end
over: BOOLEAN deferred end
setup deferred end
step do end
tear_down deferred end
end
```
Simple Concurrent Object-Oriented Programming

CACM (1993) & Object-Oriented Software Construction, 1997

Prototype implementation at ETH (since 2007); production version 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
What we write in sequential code

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

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

invariant balance >= 0
A better version

```plaintext
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!

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

\[
\text{require} \quad \text{source.balance} \geq \text{value}
\]

\[
\text{do} \quad \text{source.withdraw (value)} \quad \text{target.deposit (value)}
\]

\[
\text{ensure} \quad \text{source.balance} = \text{old source.balance} - \text{value} \quad \text{target.balance} = \text{old target.balance} + \text{value}
\]

\[
\text{end}
\]

\[
\text{...}
\]

\[
\text{invariant} \quad \text{balance} \geq 0
\]
Dining philosophers

Listing 4.33: Variables for Tanenbaum’s solution

```python
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

```python
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()
```
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
put \((b: \text{QUEUE} [G]; v: G)\)

-- Store \(v\) into \(b\).

require

not b.is_full

do

... 

ensure

not b.is_empty

b.item = v

end

my_queue: QUEUE [T]

...

if not my_queue.is_full then

put (my_queue, \(t\))

end
Reasoning about objects: sequential

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

\{\text{Pre}_r\} \quad x.r(a) \quad \{\text{Post}_r\}

Priming represents actual-formal argument substitution

Only $n$ proofs if $n$ exported routines!
In a concurrent context

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

Client 1

Client 2

Client 3

$r1$

$r2$

$r3$

No overlapping!

{INV and Pre$_r$} body$_r$ {INV and Post$_r$}

{Pre$_r'$} x.$r$ (a) {Post$_r'$}
SCOOP rules

- One processor per object: “handler”

- At most one feature (operation) active on an object at any time
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**
- ...
Feature call: sequential

\[ x.r(a) \]

Client

\[ \text{previous} \]

\[ x.r(a) \]

\[ \text{next} \]

Processor

Supplier

\[ r(x : A) \]

\[ \text{do} \]

\[ \ldots \]

\[ \text{end} \]
Feature call: asynchronous

Client

$\text{previous}
\begin{align*}
x \cdot r(a) \\
\text{next}
\end{align*}
\text{Client's handler}

Supplier

$\begin{align*}
r(x : A) \\
do \\
\text{...} \\
\text{end}
\end{align*}
\text{Supplier's handler}$
To wait or not to wait:

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

Difference must be captured by syntax:

- \( x: T \)
- \( x: \text{separate } T \)-- Potentially different processor

Fundamental semantic rule: \( x.r(a) \) waits for non-separate \( x \), doesn’t wait for separate \( x \).
```
put (b: QUEUE [G]; v: G)
-- Store v into b.
require
not b.is_full
do
...end
ensure
not b.is_empty
b.item = v
end

my_queue: QUEUE [T]
...
if not my_queue.is_full then
  put (my_queue, t)
end
```
Synchronization rule

A call with separate arguments waits until:
- The corresponding objects are all available
- Preconditions hold

It will hold the arguments for the duration of the routine’s execution

“Separate call”:

\[ x.f(a) \quad -- \text{where } a \text{ is separate} \]
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
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

...
Resynchronization

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

The client will wait only when it needs to:

\[
x.f \\
x.g(a) \\
y.f \\
\ldots
\]

\[
\text{value} := x.\text{some\_query}
\]
Refined proof rule (partial correctness)

\[
\{\text{INV} \land \text{Pre}_r (x)\} \text{ body}_r \{\text{INV} \land \text{Post}_r (x)\}
\]

\[
\{\text{Pre}_r (a^{\text{cont}})\} \ x.\ r (a) \ {\text{Post}_r (a^{\text{cont}})}
\]

Hoare-style sequential reasoning

Controlled expressions are locked by current processor
Elevator example architecture

For maximal concurrency, all objects are separate
Two topics

1. Concurrent Programming Is Easy

2. Developments at the Chair of Software Engineering: an overview
Funding

ETH
Swiss National Science Foundation
Hasler Stiftung
Gebert-Ruf Stiftung
Verification As a Matter of Course

- Inter. prover
- Sep. prover
- Auto. prover
- Alias analysis
- Invariant inference
- AutoFix

Arbiter

Suggestions

Programmer

EVE

Test execution

Test case generation

AutoTest

Suggestions

Test results
Unifying framework: Eiffel

Method and language
Language is ECMA- and ISO-standardized

Full object-oriented model
Built-in contracts

Void safety (no null pointer dereferencing)

Extensive development environment (EiffelStudio)
Push-button testing: AutoTest

Now part of EiffelStudio

- **Test generation:**
  - Fully automatic generation of test cases
  - Oracles are contracts
  - Integration with regression testing
  - Found hundreds of bugs in libraries & apps

- **Test extraction:** generate tests from failures

- **Manual testing** (à la JUnit)
Faults found per second

\[
f_c(t)
\]

Time

Faults

Number of bugs

0  2  4  6  8  10  12  14  16  18

Time

0  10  20  30  40  50  60  70
Proofs

Built-in proof tools, built on Microsoft's Boogie

Separation logic prover (Stephan van Staden, Cristiano Calcagno)
class STACK [G] feature {SPECIFICATION}

model: MML_SEQUENCE

feature
push (x: G)
  require
    not full
  do
    ...
  ensure
    top = old x
    count = old count + 1
    model = old model + [x]
  end
end

Sequence concatenation
Invariant generation

Static and dynamic approaches

- **Statically**: postcondition weakening

- **Dynamically**: learn from test runs, in connection with AutoTest (and using Michael Ernst’s Daikon)
### Invariant generation on standard examples

<table>
<thead>
<tr>
<th>Procedure</th>
<th>LOC</th>
<th># Candidates</th>
<th># Invariants</th>
<th># Relevant (%)</th>
<th>Time (sec.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array Partitioning (v1)</td>
<td>58 (22)</td>
<td>38</td>
<td>9</td>
<td>3 (33%)</td>
<td>93</td>
</tr>
<tr>
<td>Array Partitioning (v2)</td>
<td>68 (40)</td>
<td>45</td>
<td>2</td>
<td>2 (100%)</td>
<td>205</td>
</tr>
<tr>
<td>Array Stack Reversal</td>
<td>147 (34)</td>
<td>134</td>
<td>4</td>
<td>2 (50%)</td>
<td>529</td>
</tr>
<tr>
<td>Array Stack Reversal (annotated)</td>
<td>147 (34)</td>
<td>134</td>
<td>6</td>
<td>4 (67%)</td>
<td>516</td>
</tr>
<tr>
<td>Bubblesort</td>
<td>69 (29)</td>
<td>14</td>
<td>2</td>
<td>2 (100%)</td>
<td>65</td>
</tr>
<tr>
<td>Coincidence Count</td>
<td>59 (29)</td>
<td>1351</td>
<td>1</td>
<td>1 (100%)</td>
<td>4304</td>
</tr>
<tr>
<td>Dutch National Flag</td>
<td>77 (43)</td>
<td>42</td>
<td>10</td>
<td>2 (20%)</td>
<td>117</td>
</tr>
<tr>
<td>Dutch National Flag (annotated)</td>
<td>77 (43)</td>
<td>42</td>
<td>12</td>
<td>4 (33%)</td>
<td>122</td>
</tr>
<tr>
<td>Max of Array (v1)</td>
<td>27 (17)</td>
<td>13</td>
<td>1</td>
<td>1 (100%)</td>
<td>30</td>
</tr>
<tr>
<td>Max of Array (v2)</td>
<td>27 (17)</td>
<td>7</td>
<td>1</td>
<td>1 (100%)</td>
<td>16</td>
</tr>
<tr>
<td>Plateau</td>
<td>53 (29)</td>
<td>31</td>
<td>6</td>
<td>3 (50%)</td>
<td>666</td>
</tr>
<tr>
<td>Sequential Search (v1)</td>
<td>34 (26)</td>
<td>45</td>
<td>9</td>
<td>5 (56%)</td>
<td>120</td>
</tr>
<tr>
<td>Sequential Search (v2)</td>
<td>29 (21)</td>
<td>24</td>
<td>6</td>
<td>6 (100%)</td>
<td>58</td>
</tr>
<tr>
<td>Shortest Path (annotated)</td>
<td>57 (44)</td>
<td>23</td>
<td>2</td>
<td>2 (100%)</td>
<td>53</td>
</tr>
<tr>
<td>Stack Search</td>
<td>196 (49)</td>
<td>102</td>
<td>3</td>
<td>3 (100%)</td>
<td>300</td>
</tr>
<tr>
<td>Sum of Array</td>
<td>26 (15)</td>
<td>13</td>
<td>1</td>
<td>1 (100%)</td>
<td>44</td>
</tr>
<tr>
<td>Welfare Crook</td>
<td>53 (21)</td>
<td>20</td>
<td>2</td>
<td>2 (100%)</td>
<td>586</td>
</tr>
</tbody>
</table>
Alias calculus

Goal: provide automatic detection of possible aliasing, allowing sound proofs of programs involving pointers

\[
\begin{align*}
- & \quad \text{Set difference} \\
\text{a } & \Downarrow E = a - A \times E \\
& \quad \text{-- i.e. a deprived of all pairs} \\
& \quad \text{-- involving an element of E} \\
a & / x = \{ y : E | (y = x) \lor [x, y] \in a \} \\
& \quad \text{-- i.e. all elements aliased to x in a, plus} \\
& \quad \text{x itself} \\
a \uparrow \text{skip} & = a \quad /12/ \\
a \uparrow (p ; q) & = (a \uparrow p) \uparrow q \quad /13/ \\
a \uparrow \text{(forget } x) & = a \Downarrow \{ x \} \quad /14/ \\
a \uparrow \text{(create } x) & = a \Downarrow \{ x \} \quad /15/ \\
a \uparrow \text{(cut } x, y) & = a - x, y \quad /16/ \\
a \uparrow (x := y) & = a [x : y] \quad /17/ \\
a [x : y] & = \ \text{given } b \uparrow \{ x \} \text{ then} \\
& \quad \begin{array}{c}
\neg \text{ b } \cup (\{ x \} \times (b / y)) \\
\text{end}
\end{array} \quad /18/ \\
a [x : y] & = \ldots ((a [x_1 : y_1]) [x_2 : y_2]) \ldots [x_n : y_n] \quad /20/ \\
& \quad \text{-- For lists x and y}
\end{align*}
\]
Alias calculus

```
a \triangleright \text{then } p \ \text{else } q \ \text{end} = (a \triangleright p) \cup (a \triangleright q) 
```

```
a \triangleright \text{loop } p \ \text{end} = t_N
-- For the first \( N \) such that \( t_N = t_{N+1} \),
-- with \( t_0 = a \) and \( t_{n+1} = t_n \cup (t_n \triangleright p) \).
```

```
a \triangleright pr
-- For a program \( pr \) of main program \( \text{Main} \)
```

```
a \triangleright \text{call } r \ (l) = a \ [r^*: l] \triangleright r
-- Where is the list of formal arguments of \( r \)
```

```
\text{Current}_e = e
```

```
e . \text{Current} = e
```

```
x . x' = \text{Current}
```

```
x' . x = \text{Current}
```

```
x . x' . e = e
```

```
x' . x . e = e
```

```
a \triangleright \text{call } x . r \ (l) = x \cdot ((x' . a \ [r^*: x' . l]) \triangleright r) \ldots x \cdot r^*
```

```
a \triangleright \text{call } x . r = x \cdot ((x' . a) \triangleright r)
```

43
AutoFix: use data mining techniques and contracts to propose corrections after detecting a failure
Verification As a Matter of Course

Inter. prover → Programmer

Sep. prover → Suggestions

Auto. prover → Invariant inference

Alias analysis → AutoFix

Arbiter

Suggestions

EVE

Test execution

Test case generation

AutoTest

Test results
Further developments of the SCOOP model

- Theory (structured operational semantics)
- Deadlock prevention and detection
- Robotics applications (Ganesh Ramanathan)
An application: hexapod robot

Distributed control
Load sensing

Centralised control
Balance sensing

Ganesh Ramanathan, Sebastian Nanz, Benjamin Morandi, Scott West
The Tripod gait

Alternating protraction and retraction of tripod pairs

- Begin protraction only if partner legs are down
- Depress legs only if partner legs have retracted
- Begin retraction when partner legs are up
Introductory programming: “Inverted Curriculum” (outside-in approach), since 2003

Distributed Software Engineering: DOSE course
(Distributed and Outsourced Software Engineering)
Distributed student project (ETH, U. Zurich, Politecnico di Milano, Ukraine, Russia, Hungary, Argentina, Korea, Vietnam, ...)

Michela Pedroni, Karine Arnout, Manuel Oriol & many students
Distributed software development

CloudStudio: software development on the cloud

Sought: a new approach to configuration management
For more

http://se.ethz.ch

http://www.eiffel.com