The SCOOP model

Bertrand Meyer

Credits to: Piotr Nienaltowski, Patrick Eugster

LASER Summer School, 2005

© Bertrand Meyer, 2005
if not \texttt{that\_box.is\_full} then
\begin{verbatim}
    put (that_box, t1)
\end{verbatim}
end

\hspace{1cm} -- Another client can break in!

\begin{itemize}
  \item \texttt{put (q: BUFFER [T]; x: T)}
  \hspace{1cm} -- Store $x$ into $q$. \\
  \hspace{1cm} require \\
  \hspace{1cm} \texttt{not q.is\_full}
  \hspace{1cm} do \\
  \hspace{1cm} \hspace{1cm} ... \\
  \hspace{1cm} \hspace{1cm} ensure \\
  \hspace{1cm} \hspace{1cm} \texttt{not q.is\_empty}
  \hspace{1cm} end
\end{itemize}
Basic goal

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

<table>
<thead>
<tr>
<th>Feature</th>
<th>“Structured programming”</th>
<th>“Object technology”</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use higher-level abstractions</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Helps avoid bugs</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Transfers tasks to implementation</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Lets you do stuff you couldn’t before</td>
<td>NO</td>
<td>✓</td>
</tr>
<tr>
<td>Removes restrictions</td>
<td>NO</td>
<td>✓</td>
</tr>
<tr>
<td>Adds restrictions</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Has well-understood math basis</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Doesn’t require understanding that basis</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Permits less operational reasoning</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
Reasoning about objects

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

\text{Priming: substitution of actuals for formals}
SCOOP in a nutshell

Eiffel extension with one keyword (*separate*)

Object access is exclusive for the duration of a routine

New semantics for precondition: wait

Library mechanisms for interruption of reservation
Dining philosophers

class PHILOSOPHER inherit PROCESS
   rename
      setup as getup
   redefine step end

feature {BUTLER}
   step is
      do
         think; eat (left, right)
      end

   eat (l, r. separate FORK) is
      -- Eat, having grabbed l and r.
      do ... end

   end
The world is not just sequential any more

Multithreading

Internet-based applications

Distribution

Pervasive computing

Web services

Everyone wants to do it. Many say they’re doing it. Many are doing it. Those doing it are not doing it very well!
Then and now

Sequential programming:
Used to be messy
Still hard but:
- Structured programming
- Data abstraction & object technology
- Design by Contract
- Genericity, multiple inheritance
- Architectural techniques
Switch from operational reasoning to logical deduction (e.g. invariants)

Concurrent programming:
Used to be messy
Still messy
Example: threading models in most popular approaches
Development level: ca. 1968
Only understandable through operational reasoning
What O-O brought to sequential programming

Structuring concept: the class

- Module-type fusion
- Information hiding
- Multiple inheritance
- Genericity
- Polymorphism and dynamic binding
- Design by Contract™

Computed concept: the object

- Modeling power
- Dynamic allocation
- Automatic memory management

x.r (a)
This mechanism

**SCOOP: Simple Concurrent Object-Oriented Programming**

First iteration 1990

*CACM, 1993*

Object-Oriented Software Construction, 2nd edition, 1997

Prototype implementation at Eiffel Software, 1995

Prototypes by others

No being done for good at ETH, Hasler foundation funding, also ETH and Microsoft ROTOR project
Can object technology help?

“Objects are naturally concurrent” (Milner)

Many attempts, often based on (self-contradictory) notion of “Active objects”

Often lead to “Inheritance anomaly”

None widely accepted

In practice: low-level mechanisms on top of O-O language
Processes?

```plaintext
defered class PROCESS feature
    live
        -- General structure with variants.
        do
            from setup until over loop
                step
            end
        finalize
    end
feature {NONE}
    setup deferred end
    over: BOOLEAN deferred end
    step deferred end
    finalize deferred end
end
```
Can object technology help?

Impedance mismatch:

- **O-O**: high-level abstraction mechanisms

- Concurrency: semaphores, locks, suspend, manual exclusion, sharing...
Explore how the concepts of object technology can help us tackle the challenge of concurrent applications, as they did for sequential programming.
Feature call

\[ x : CX \]

\[ x.r(a) \]

**Client**

\textit{previous\_instruction}

\[ x.r(a) \]

\textit{next\_instruction}

**Supplier (CX)**

\[ r(a:A) \]

- require \( a \neq \text{Void} \)
- ensure not \( a.is\_empty \)
- end
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
- Web site
- Process
- Thread
- AppDomain (.NET) ...

Will be mapped to computational resources
Handling rule

All calls on an object are executed by the object’s handler
Reasoning about objects

\{Pre_r \textbf{and} INV\} \; body_r \; \{Post_r \textbf{and} INV\}

\hline
\{Pre'_r\} \times_r (a) \; \{Post'_r\}
Reasoning about objects

Only $n$ proofs if $n$ exported routines!

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

\[
\{\text{Pre}_r'\} \ x.r \ (a) \quad \{\text{Post}_r'\}
\]
In a concurrent context

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

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

$\{\text{Pre}_r '\} \times .r(a) \quad \{\text{Post}_r '\}$

Client 1, $r1$ \hspace{2cm} Client 2, $r2$ \hspace{2cm} Client 3, $r3$
Mutual exclusion rule

At most one feature may execute on any one object at any one time
Feature call: sequential

\[ x : CX \]

\[ x.r(a) \]

Client

```
previous_instruction

x.r(a)

next_instruction
```

Supplier (CX)

```
r(a : A)

require
a /= Void

ensure
not a.is_empty

end
```
Feature call: asynchronous

\[ x.r(a) \]

\( x: \) separate \( CX \)

Client

\( \text{previous}_\text{instruction} \)

\( x.r(a) \)

\( \text{next}_\text{instruction} \)

Client processor

Supplier (\( CX \))

\( r(a: A) \)

require \( a \neq \text{Void} \)

ensure not \( a.is\_empty \)

Supplier processor
Separateness rule

Calls to non-separate objects are synchronous

Call to separate objects are asynchronous
The fundamental difference

To wait or not to wait:
- If same processor, synchronous
- If different processor, asynchronous

Difference must be captured by syntax:

- \( x: CX \)
- \( x: \) separate \( CX \)
Feature call: asynchronous

\[ x \cdot r(a) \]

\( x \): separate \( CX \)

Client

\[ \text{previous\_instruction} \]

\[ x \cdot r(a) \]

\[ \text{next\_instruction} \]

Supplier (\( CX \))

\[ r(a; A) \text{ is} \]

require \( a \neq \text{Void} \)

ensure not \( a \ \text{is\_empty} \)

end

Client processor

Supplier processor
What does “separate” mean?

Does not specify processor

Simply indicates that it’s “elsewhere”
Consistency

A **traitor** is a variable that denotes a separate object and is not declared as separate.
Consistency 1

May not attach separate to non-separate

In

\[ x := y \]

or

\[ r(y) \quad -- \text{with formal argument } x \]

if \( x \) is declared \textbf{separate}, so must \( x \).
Separateness attachment rule

The target of an attachment with a separate target must be separate.

Attachment: assignment or argument passing
Consistency 2

Client:

```ruby
class C feature
  a: SOME_TYPE
  sep: separate B
  sep.p(a)
end
```

Supplier:

```ruby
class B feature
  p(a: SOME_TYPE)
    is do ... end
end
```
Consistency

Client:

class C feature
    a: SOME_TYPE
    sep: separate B
    sep.p (a)
end

Supplier:

class B feature
    p (a: separate SOME_TYPE)
        is do ... end
end
Separateness consistency rule

For any reference actual argument in a separate call, the corresponding formal argument must be declared as separate.

Separate call: \( a.f (...) \) where \( a \) is separate
If no access control

\[ x : \text{separate } CX \]

\[ \ldots \]

\[ x.r(a) \]

\[ y := x.f \]
If no access control

\[ \text{my\_stack} : \text{separate STACK}[T] \]

\[ \ldots \]

\[ \text{my\_stack}.\text{push}(a) \]

\[ y := \text{my\_stack}.\text{top} \]
Access control policy

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

\[
\text{put} (b: \text{separate} \ STACK [T]; \text{value: } T) \text{ is }
\]

-- Push value on top of b.

\[
\text{do }
\]

\[
\ b.\text{push} (\text{value})
\]

\[
\text{end}
\]
Access control policy

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

\[
\text{put} (b: \text{separate BUFFER}[T]; \text{value}: T) \text{ is}
\]

\[
\quad \text{-- Store value into } b.
\]

\[
\begin{align*}
& \text{do} \\
& \quad b.\text{put} (\text{value}) \\
& \text{end}
\end{align*}
\]

To use separate object:

\[
\text{my\_buffer: separate BUFFER}[\text{INTEGER}] \\
\text{create my\_buffer} \\
\text{store (my\_buffer, 10)}
\]
Separate argument rule

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

Separate call: \( a.f (...) \) where \( a \) is separate
A call \( r(a, b, c) \) with

\[
r(x: \text{separate } T; \quad y: U; \quad z: \text{separate } V)
\]

will not proceed until the objects attached to \( x \) and \( z \) are available.
Wait rule

A routine call with separate arguments will execute when all corresponding objects are available and hold them exclusively for the duration of the routine.

Separate call: $a.f (...)$ where $a$ is separate.
Dining philosophers

class PHILOSOPHER inherit PROCESS
    rename
        setup as getup
    redefine step end

feature {BUTLER}
    step is
        do
            think; eat(left, right)
        end

    eat(l, r: separate FORK) is
        -- Eat, having grabbed l and r.
        do ... end

end
If no waiting is desired

With

\[ r(x : ? \text{ separate } T) \]

A call \( r(a) \) will not wait.

Quiz: what can \( r \) do with \( x \)?
Contracts in Eiffel

```eiffel
class store (buffer : BUFFER [INTEGER]; value: INTEGER) is
  -- Store value into buffer.
  require
    not buffer.is_full
    value > 0
  do
    buffer.put (value)
  ensure
    not buffer.is_empty
  end

  ...
  store (my_buffer, 10)
```

Precondition
store (buffer: separate BUFFER [INTEGER]; value: INTEGER)

is
  -- Store value into buffer.
require
  not buffer.is_full
  value > 0
do
  buffer.put (value)
ensure
  not buffer.is_empty
end

... store (my_buffer, 10)

If buffer is separate, ..

On separate target, precondition becomes wait condition
Contracts

Supplier:

\[
\text{store (b: BUFFER [T]; value: T) is}
\]

\[
\quad \text{-- Store value into b.}
\]

\[
\quad \text{require}
\]

\[
\quad \quad \text{not b.is_full}
\]

\[
\quad \quad \text{value > 0}
\]

\[
\quad \text{do}
\]

\[
\quad \quad b.put (\text{value})
\]

\[
\quad \text{ensure}
\]

\[
\quad \quad \text{not b.is_empty}
\]

\[
\quad \text{end}
\]

\[
\quad \ldots
\]

Client:

\[
\text{if not my_buffer.is_full}
\]

\[
\text{then}
\]

\[
\text{store (my_buffer, x)}
\]

\[
\text{end}
\]
Contracts under concurrency?

Client:

```plaintext
if not my_buffer.is_full then
  store (my_buffer, x)
end
```

Supplier:

```plaintext
store (b: BUFFER [T]; value: T) is
  -- Store value into b.
  require
    not b.is_full
    value > 0
  do
    b.put (value)
  ensure
    not b.is_empty
end...
```
What happens to preconditions?

Precondition on separate target becomes **wait condition** (instead of correctness condition)

This becomes the basic synchronization mechanism
A separate precondition causes the client to wait

Separate precondition: \( a \cdot \text{condition} (\ldots) \)
where \( a \) is separate
Full synchronization rule

A call with a separate argument waits until:

- Object is available
- Separate precondition holds

\[ x.f(a) \]

where \( a \) is separate
Bounded buffer usage

```plaintext
buff: BUFFER_ACCESS[MESSAGE]
my_buffer:
  BOUNDED_BUFFER[MESSAGE]

create my_buffer
create buff.make (my_buffer)

buff.store(my_buffer, my_message)
...
buff.store (my_buffer, her_message)
...
my_query := buff.item (my_buffer)

store
  (b: separate BUFFER [G];
   value: G)
    -- Store value into b.
require
  not buffer.is_full
  value > 0
do
  buffer.put (value)
ensure
  not buffer.is_empty
end
```
Resynchronization

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

The client will wait only when it needs to:

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

\( value := x\text{.some_query} \)
Resynchronization rule

Clients wait for resynchronization on queries
Example: class *PROCESS*

defered class
   
   PROCESS

feature -- Status report
   
   over: BOOLEAN is
      -- Must execution terminate now?
      deferred end

feature -- Basic operations
   
   setup is
      -- Prepare to execute process (default: nothing).
      do end

   step is
      -- Execute basic process operations.
      deferred end
PROCESS

wrapup is
  -- Execute termination operations (default: nothing).
  do end

feature -- Process behavior

live is
  -- Perform process lifecycle.
  do
    from setup until over loop
      step
    end
    wrapup
  end
end
Example: Dining philosophers

class PHILOSOPHER inherit PROCESS
  rename setup as getup
  redefine step end

feature {BUTLER}
  step is
    do
      think;  eat (left, right)
    end

  eat (l, r: separate FORK) is
    -- Eat, having grabbed l and r.
    do ... end
end
SCOOP multithreaded elevators
Elevator example architecture

For maximal concurrency, all objects are separate
Dynamic diagram

Scenario: Pressing the cabin button to move the elevator

1. Cabin button calls `elevator.accept (target)`
2. Elevator calls `engine.move (floor)`
3. Engine calls `gui_main_window.move_elevator (cabin_number, floor)`
4. Engine calls `elevator.record_stop (position)`
Class BUTTON

separate class

BUTTON

feature

target: INTEGER

end
Class **CABIN_BUTTON**

```plaintext
separate class CABIN_BUTTON inherit BUTTON

feature
  cabin : ELEVATOR

  request
    -- Send to associated elevator a request to stop on level \textit{target}.
    do
      actual_request (cabin)
    end

  actual_request (e : ELEVATOR)
    -- Get hold of \textit{e} and send a request to stop on level \textit{target}.
    do
      e.accept (target)
    end
end
```
Class **ELEVATOR**

separate class **ELEVATOR** feature \{BUTTON, DISPATCHER\}

```plaintext
accept (floor: INTEGER)
  -- Record and process a request to go to floor.
  do
    record_stop (floor)
    if not moving then process_request end
  end

feature \{MOTOR\}

record_stop (floor: INTEGER)
  -- Record information that elevator has stopped on floor.
  do
    moving := False ; position := floor ; process_request
  end
```
Class **ELEVATOR**

```plaintext
feature {NONE} -- Implementation

process_request

    -- Handle next pending request, if any.

local  floor: INTEGER do

    if not pending.is_empty then
        floor := pending.item; actual_process(puller, floor)
        pending.remove
    end

end

end
```
actual_process (m : MOTOR; floor : INTEGER)
   -- Handle next pending request, if any.
   do
      moving := True ; m.move (floor)
   end

feature {NONE} -- Implementation
   puller : MOTOR ; pending : QUEUE [INTEGER]
end
Class \textit{MOTOR}

\begin{verbatim}
separate class \textit{MOTOR} feature \{\textit{ELEVATOR}\}
  move (floor: INTEGER)
    -- Go to \textit{floor}; once there, report.
    do
      gui_main_window . move_elevator (cabin_number, floor)
      signal_stopped (cabin)
    end
  signal_stopped (e: \textit{ELEVATOR})
    -- Report that elevator \textit{e} stopped on level \textit{position}.
    do
      e . record_stop (position)
    end
feature \{\textit{NONE}\}
  cabin: \textit{ELEVATOR}; position: INTEGER    -- Current floor level.
  gui_main_window: \textit{GUI\_MAIN\_WINDOW}
end
\end{verbatim}
A digression

Exception handling
Causes of exceptions

Void call ($x.f$ with no object attached to $x$)

Operating system signal: arithmetic overflow, no more memory, interrupt ...

Assertion violation (if contracts are being monitored)
Exceptions

An exception is an “abnormal case” occurring program execution, causing a disruption of the default flow of control.
How to use exceptions?

Two opposite styles:

- Exceptions as a control structure: Use an exception to handle all cases other than the most favorable ones (e.g. a key not found in a hash table triggers an exception)

- Exceptions as a technique of last resort
Exception vocabulary

- "Raise", "trigger" or "throw" an exception
- "Handle" or "catch" an exception
Java exceptions

Exceptions are objects, descendants of Throwable:
Java: raising an exception

**Instruction:**

```java
throw my_exception
```

The enclosing routine should be of the form

```java
myroutine (...) throws my_exception {
    ...
    if abnormal_condition
        throw my_exception;
}
```
Java: handling an exception

```java
try {
    instruction_1;
    instruction_2;
    ...
    instruction_n;
}

catch (Expected_exception_type e) {
    handling_code
}

(Possible “finally” clause to complete both cases)
```
Exception handling

The need for exceptions arises when a contract is broken.

Two concepts:

- **Failure**: a routine, or other operation, is unable to fulfill its contract.
- **Exception**: an undesirable event occurs during the execution of a routine — as a result of the failure of some operation called by the routine.
The original strategy

\[ r(...) \text{ is} \]

\begin{align*}
&\text{require} \\
&\quad \ldots \\
&\text{do} \\
&\quad op_1 \\
&\quad op_2 \\
&\quad \ldots \\
&\quad op_i \\
&\text{ensure} \\
&\quad \ldots \\
&\text{end}
\end{align*}

Fails, triggering an exception in \( r \) (\( r \) is \textit{recipient} of exception).
Handling exceptions properly

Safe exception handling principle:

There are only two acceptable ways to react for the recipient of an exception:

- Concede failure, and trigger an exception in the caller (*Organized Panic*)
- Try again, using a different strategy (or repeating the same strategy) (*Retrying*)

(Rare third case: false alarm)
How not to do it

(From an Ada textbook)

sqrt (x : REAL) return REAL is
begin
  if x < 0.0 then
    raise Negative;
  else
    normal_square_root_computation;
  end
exception
  when Negative =>
    put ("Negative argument");
    return;
  when others => ...
end; -- sqrt
The call chain

Routine call
Exception mechanism

Two constructs:
- A routine may contain a rescue clause.
- A rescue clause may contain a retry instruction.

A rescue clause that does not execute a retry leads to failure of the routine (this is the organized panic case).
Transmitting over an unreliable line (1)

```
Max_attempts: INTEGER is 100

attempt_transmission (message: STRING) is
    -- Transmit message in at most
    -- Max_attempts attempts.
    local
        failures: INTEGER
do
    unsafe_transmit (message)
rescue
    failures := failures + 1
    if failures < Max_attempts then
        retry
    end
end
```
Transmitting over an unreliable line (2)

Max_attempts: INTEGER is 100
failed: BOOLEAN

attempt_transmission (message: STRING) is
   -- Try to transmit message;
   -- if impossible in at most Max_attempts
   -- attempts, set failed to true.
   local
      failures: INTEGER
   do
      if failures < Max_attempts then
         unsafe_transmit (message)
      else
         failed := True
      end
   rescue
      failures := failures + 1
      retry
   end
If no exception clause (1)

Absence of a rescue clause is equivalent, in first approximation, to an empty rescue clause:

\[ f(...) \text{ is} \]
\[ \text{do} \]
\[ \text{...} \]
\[ \text{end} \]

is an abbreviation for

\[ f(...) \text{ is} \]
\[ \text{do} \]
\[ \text{...} \]
\[ \text{rescue} \]
\[ \text{-- Nothing here} \]
\[ \text{end} \]

(This is a provisional rule; see next.)
The correctness of a class

(1-n) For every exported routine \( r \):

\[ \{ \text{INV and Pre}_r \} \text{ do } \{ \text{Post}_r \text{ and INV} \} \]

(1-m) For every creation procedure \( cp \):

\[ \{ \text{Pre}_{cp} \} \text{ do } \{ \text{Post}_{cp} \text{ and INV} \} \]
Exception correctness

For the normal body:

\{INV \text{ and } Pre_r\} \ 	ext{do}\ r \ \{Post_r \text{ and } INV\}

For the exception clause:

\{ ??? \} \ 	ext{rescue}\ r \ \{ ??? \}
Exception correctness

For the normal body:

\{\text{INV and Pr}_r\} \text{ do } \{\text{Post}_r \text{ and INV}\}

For the exception clause:

\{\text{True}\} \text{ rescue } \{\text{INV}\}
If no exception clause (2)

Absence of a rescue clause is equivalent to a default rescue clause:

\[
\text{f(...) is do ... end is an abbreviation for}
\]

\[
\text{f(...) is do ... rescue default_rescue end}
\]

The task of \textit{default_rescue} is to restore the invariant.
For finer-grain exception handling

Use class \texttt{EXCEPTIONS} from the Kernel Library.

Some features:

\begin{itemize}
  \item \texttt{exception} (code of last exception that was triggered).
  \item \texttt{assertion\_violation}, etc.
  \item \texttt{raise ("exception\_name")}
\end{itemize}
SCOOP for real-time systems

Possibility to execute the request of a VIP client while preempting the current client
- Duels
- Priorities

Timing assertions
- Maximal (and minimal) execution time
- Timeouts on actions
- Periodic and aperiodic activities
Duels

Problem: Impatient client (*challenger*) wants to snatch object from another client (*holder*)

Can't just interrupt holder, service challenger, and resume holder: would produce inconsistent object.

But: can cause exception, which will be handled safely.
Interrupts?

Can we snatch shared object from its current holder?

Execute $\text{holder}.r(b)$ where $b$ is separate

Another object executes $\text{challenger}.s(b)$

Normally, $\text{challenger}$ would wait

What if $\text{challenger}$ is impatient?
## Semantics of duels

<table>
<thead>
<tr>
<th>challenger →</th>
<th>holder</th>
<th>normal_service</th>
<th>demand</th>
<th>demand_if_possible</th>
</tr>
</thead>
<tbody>
<tr>
<td>retain</td>
<td></td>
<td>challenger waits</td>
<td>exception in challenger</td>
<td>challenger waits</td>
</tr>
<tr>
<td>yield</td>
<td></td>
<td>challenger waits</td>
<td>exception in Holder</td>
<td>exception in holder</td>
</tr>
</tbody>
</table>
## Semantics of duels with timeouts

<table>
<thead>
<tr>
<th>challenger →</th>
<th>normal_service</th>
<th>demand_with_timeout (time)</th>
<th>demand_if_possible_with_timeout (time)</th>
</tr>
</thead>
<tbody>
<tr>
<td>↓ holder</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>retain</td>
<td>challenger waits</td>
<td>exception in challenger</td>
<td>challenger waits</td>
</tr>
<tr>
<td>yield</td>
<td>challenger waits</td>
<td>exception in Holder</td>
<td>exception in holder</td>
</tr>
</tbody>
</table>
# Semantics of duels with timeouts

<table>
<thead>
<tr>
<th>challenger →</th>
<th>normal_service</th>
<th>demand_with_timeout (time)</th>
<th>demand_if_possible_with_timeout (time)</th>
</tr>
</thead>
<tbody>
<tr>
<td>↓ holder</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>retain_after_timeout (time)</td>
<td>challenger waits</td>
<td>exception in challenger</td>
<td>challenger waits</td>
</tr>
<tr>
<td>yield_after_timeout (time)</td>
<td>challenger waits</td>
<td>exception in Holder</td>
<td>exception in holder</td>
</tr>
</tbody>
</table>
A watchdog mechanism

defered class TIMED inherit
    CONCURRENCY
feature {NONE}
    failed : BOOLEAN; alarm : WATCHDOG
    timed_action (t : REAL)
        -- Execute action, interrupting after t secs if not complete.
        -- If interrupted before completion, set failed to true.
        do
            set_alarm (t); unset_alarm (t); failed := False
        rescue
            if is_concurrency_interrupt then failed := True end
        end
Class **TIMED** (continued)

\[
\textit{set\_alarm} \ (t: \text{REAL})
\]

\[
\quad -- \text{Set alarm to interrupt current object after } t \text{ seconds.}
\]

\[
\quad \text{do}
\]

\[
\quad \quad -- \text{Create alarm if necessary:}
\]

\[
\quad \quad \text{if } \text{alarm} = \text{Void} \text{ then create alarm end}
\]

\[
\quad \quad \text{yield; actual\_set(} \text{alarm, } t\text{); retain}
\]

\[
\quad \text{end}
\]

\[
\textit{unset\_alarm} \ (t: \text{REAL})
\]

\[
\quad -- \text{Remove the last alarm set.}
\]

\[
\quad \text{do demand; actual\_unset(} \text{alarm); wait\_turn end}
\]

\[
\text{action}
\]

\[
\quad -- \text{Action to be performed under watchdog control}
\]

\[
\text{deferred end}
\]
separate class
    WATCHDOG
feature {TIMED}
    set(caller: separate TIMED; t: REAL)
        -- After t seconds, interrupt caller;
        -- if interrupted before, terminate silently.
        local
            interrupted: BOOLEAN
        do
            if not interrupted then
                wait(t); demand; caller.stop; wait_turn
            end
        rescue
            if is_concurrency_interrupt then
                interrupted := True; retry
            end
        end
end
Class WATCHDOG (end)

unset is
    -- Remove alarm (empty action to let client interrupt set).
    do -- Nothing end
feature {NONE}
    early_termination: BOOLEAN
end
feature \{NONE\} -- Actual access to watchdog

  actual_set (a: WATCHDOG, t: REAL)

    -- Start up a to interrupt current object after t seconds.
    do
      a.set(t)
    end

... Procedure actual_unset similar, left to the reader ...

feature \{WATCHDOG\} -- The interrupting operation

  stop

    -- Empty action to let interrupt call to timed_action
    do -- Nothing end

end
Extending duels with priorities

Tuning the duel mechanism:

holder.set_priority (50)
challenger.set_priority (100)
holder.yield
challenger.demand

If challenger.priority > holder.priority then
  holder will get an exception
  otherwise challenger will get an exception
Priorities

Extending $yield$ to specific clients (e.g. $MOTOR$)
For example $yield\,\{MOTOR\}$

$set\_priority\,(i:\,INTEGER)$
$yield\,\{MOTOR\}$ can be mapped to priorities

$yield$ could even be mapped to a special case of priority.
Timing in sequential programs

class TIMING_ASSERTION

feature -- Access

  time_now: TIME
    -- The current time now.

  max_time_duration: TIME_DURATION
    -- Maximal time duration of a feature

  min_time_duration: TIME_DURATION
    -- Minimal time duration of a feature

end -- class TIMING_ASSERTION
Two-level architecture of SCOOP

Adaptable to many environments
.NET remoting is current platform
Other examples

Watchdog: use duels

Elevator (see next)

Others in Object-Oriented Software Construction
Mapping processors to physical resources

Concurrency Control File (CCF)

create
  system
    "lincoln" (4): "c:\prog\appl1\appl1.exe"
    "roosevelt" (2): "c:\prog\appl2\appl2.dll"
    "Current" (5): "c:\prog\appl3\appl3.dll"
  end
external
  Database_handler: "jefferson" port 9000
  ATM_handler: "gates" port 8001
end
default
  port: 8001; instance: 10
end
SCOOPLI: Library for SCOOP

Library-based solution

Implemented in Eiffel for .NET
(from Eiffel Software:
EiffelStudio / ENViSioN! for Visual Studio.NET)

Aim: try out solutions without bothering with compiler issues

Can serve as a basis for compiler implementations
Original implementation

- *separate client*
- *separate supplier*

Each separate client & separate supplier handled by different processor

*Class gets separateness through multiple inheritance:*

![Diagram showing multiple inheritance]
Implementation

• Use proxy classes rather than subclasses to represent separate types
• Advantages: better correspondence with the type system

\[ X \text{ conforms to separate } X \]

• Conformance achieved through the use of conversion
• Conversion is automatic: an object is seen as non-separate to clients handled by its processor but as separate by those handled by different processors
### SCOOPLI emulation of SCOOP concepts

<table>
<thead>
<tr>
<th>SCOOP</th>
<th>SCOOPLI</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$: separate $X$</td>
<td>$x$: SEPARATE $X$</td>
</tr>
<tr>
<td>$x$: $X$ -- class $X$ is separate</td>
<td>-- SEPARATE $X$ inherits from $X$ and SEPARATE SUPPLIER</td>
</tr>
</tbody>
</table>

---

$$r(x, y)$$

-- $x$ and $y$ are separate

$$r(x: \text{separate } X; y: \text{separate } Y)$$

is

require

- not $x.is\_empty$
- $y.count > 5$
- $i > 0$ -- $i$ non-separate

$x /= \text{Void}$
do
...end

---

separate_execute ([x, y], agent $r(x, y)$, agent $r.precondition$)

$r.precondition$: BOOLEAN is
do

Result := not $x.is\_empty$ and $y.count > 5$
end

-- client class inherits from
-- class SEPARATE_CLIENT
SCOOPLI Architecture

**SEPARATE_HANDLER**: locking; checking wait conditions; scheduling of requests

**PROCESSOR_HANDLERs**: execute separate calls; implement processors

---

Inheritance

---

Client → Inheritance
Distributed execution

Processors (AppDomains) located on different machines
.NET takes care of the "dirty work"

- Marshalling
- Minimal cost of inter-AppDomain calls
Why SCOOP?

**SCOOP model**
- Simple yet powerful
- Easier and safer than common concurrent techniques, e.g. Java Threads
- Full concurrency support
- Full use O-O and Design by Contract
- Supports various platforms and concurrency architectures
- One new keyword: `separate`

**SCOOPLI library**
- SCOOP-based syntax
- Implemented on .NET
- Distributed execution with .NET Remoting
Why SCOOP?

Extend object technology with general and powerful concurrency support

Provide the industry with simple techniques for parallel, distributed, internet, real-time programming

Make programmers sleep better!
Future work & open problems

Other “handles”
Distribution and Web Services
Prevent deadlock, extend access control policy
Extend for real-time
- Duel mechanism with priorities
- Timing assertions?

Integrate with Eiffel Software compiler