A Model for Concurrent Object-Oriented Programming

Emil Sekerinski

McMaster University
Hamilton, Ontario, Canada
and
ETH Zürich, Switzerland

December 2003

Introduction

- Objects are a natural "unit" of concurrency:
  - objects can evolve independently and thus concurrently
  - method calls allow for communication and synchronization
  - creating an object potentially initiates a concurrent activity
- We present a model for active (autonomous) objects:
  - autonomous activity expressed by actions
  - synchronization expressed by guards
- Compared to mainstream languages, it simplifies the design:
  - no threads (or processors), therefore
  - no distinction between thread structure and class structure

Fish Screen Saver

```
class Fish
    attr x, y: integer
    attr up, right: boolean
    initialization
        x, y, up, right := 0, 0, true, true
    action moveUp
        when y < H and up do y := y + 1
    action moveDown
        when y > 0 and ¬up do y := y – 1
    …
    action bounceUp
        when y = H and up do up := false
    action bounceDown
        when y = 0 and ¬up do up := true
    …
end
```

- Main program:

```
var f: Fish ;
for I := 1 to 10 do
    f := new Fish
```

Bounded Buffer

```
class BoundedBuffer
    attr b: array of Object
    attr in, out, n, max: integer
    initialization (m: integer)
        in, out, n, max, b := 0, 0, 0, m, new Object [m]
    method put (x: Object)
        when n < max do
            in, b[in], n := (in + 1) mod max, x, n + 1
    method get: Object
        when n > 0 do
            out, result, n := (out + 1) mod max, b[out], n – 1
end
```

- Filtering from buffer in into out:

```
x := in.get ; if f (x) then out.put (x)
```

execution may block at calls to guarded methods

Semaphore

```
class Semaphore
    attr n: integer
    initialization (m: integer)
        n := m
    method acquire
        when n > 0 do n := n – 1
    method release
        n := n + 1
end
```

- **A semaphore** s **that allows** m **concurrent users of a resource:**

  var s: Semaphore; s := **new** Semaphore (m)

- **A user requiring semaphores** s **and** t **for a critical section:**

  s.acquire ; t.acquire ; … critical section … ; s.release ; t.release

Introducing Concurrency

```
class Doubler
    attr x: integer
    method store (u: integer)
        x := 2 ⨂ u
    method retrieve: integer
        result := x
end
```

- **Typical pattern for file operations, network transmission, web browsers, …**

```
class DelayedDoubler
    attr x: integer
    attr d: boolean
    initialization d := true
    method store (u: integer)
        x, d := u, false
    method retrieve: integer
        when d do
            result := x
    action double
        when ¬d do
            x, d := 2 ⨂ x, true
end
```

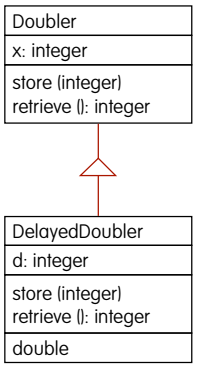
- **Objects of class** DelayedDouber **can be used wherever objects of class** Doubler **are expected.** DelayedDoubler **refines** Doubler**:**

  Doubler ⊑ DelayedDoubler

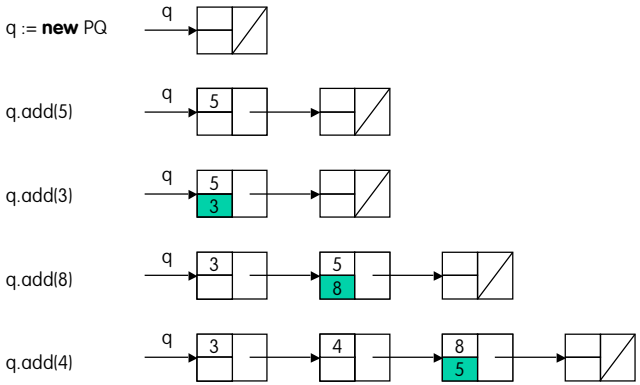
Introducing Concurrency in Subclasses

```
class Doubler
    attr x: integer
    method store (u: integer)
        x := 2 ⨂ u
    method retrieve: integer
        result := x
end

class DelayedDoubler
inherit Doubler
    attr d : boolean
    initialization d := true
    method store (u: integer)
        x, d := u, false
    method retrieve: integer
        when d do
            result := x
    action double
        when ¬d do
            x, d := 2 ⨂ x, true
end
```



Concurrent Priority Queue …



… Concurrent Priority Queue …

```
class PriorityQueue
    attr m, p: integer
    attr l: PriorityQueue
    attr a: boolean
    initialization l, a := nil, false
    method empty: boolean
        result := l = nil
    method add (e: integer)
        when ¬ a do
            if l = nil then
                begin m := e ; l := new PriorityQueue end
            else
                begin p := e ; a := true end
    action doAdd
        when a do
            begin
                if m < p then l.add (p)
                else begin l.add (m) ; m := p end ;
                a := false
            end
end
```


… Concurrent Priority Queue

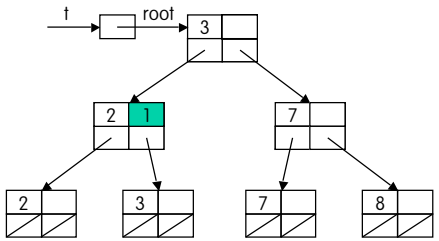
```
class PriorityQueueSpecification
    attr s: set of integer
    method empty: boolean
        result := s ≠ {}
    method add (e : integer)
        s := s ∪ {e}
    method remove: integer
        var h := min (s) ;
        begin s := s – {h} ; result := h end
end
```

- **Correctness:**

  PriorityQueueSpecification ⊑ PriorityQueue

Concurrent Leaf-oriented Trees

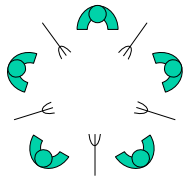
```
t := new Tree ;
t.add(3) ;
t.add(7) ;
t.add(2) ;
t.add(8) ;
t.add(1)
```



local invariant: (left = nil) = (right = nil)

global invariant: (left ≠ nil) ⇒ (left.key ≤ key) ∧ (right.key > key)

Dining Philosophers …



```
class Fork
    attr available: boolean
    initialization
        available := true
    method pickUp
        when available do
            available := false
    method putDown
        available := true
end
```

```
class Philosopher
    attr state: (thinking, hungry, eating, full)
    attr left, right: Fork
    initialization (l, r : Fork)
        state, left, right := thinking, l, r
    action gettingHungry
        when state=thinking do
            begin state := hungry ;
                left.pickup ; right.pickup ;
                state := eating
            end
    action gettingFull
        when state = eating do
            begin state := full ;
                left.putdown ; right. putdown ;
                state := thinking
            end
end
```

… Dining Philosophers

- **Main program:**

```
var fork := new Fork [5];
var philosopher := new Philosopher [5];

for i := 0 to 4 do
    fork [i] := new Fork ;
for j := 0 to 4 do
    philosopher [j] := new Philosopher (fork [j], fork [(j + 1) mod 5])
```

- **Deadlock can be avoided:**
  - **One philosopher picks up first right then left fork**
  - **Butler ensures that at most 4 philosophers are seated**
  - **Philosophers pick up both forks simultaneously.**

- **Fairness needed to avoid starvation.**

Fairness through Strong Semaphore

```
class WeakBinarySemaphore
    attr a: boolean
    initialization
        a := true
    method acquire
        when a do
            a := false
    method release
        a := true
end
```

```
class StrongBinarySemaphore
    attr a: boolean
    attr q: seq of Object
    initialization
        a, q := true, []
    method acquire (u: Object)
        begin q := q ° [u];
            when a and u = head (q) do
                a, q := false, tail (q)
    method release
        a := true
end
```

- **If continuously several users try to acquire a weak semaphore, some may be delayed indefinitely.**

- **The strong semaphore ensures a first-in first-out policy. Typical use:**

```
s.acquire (this) ; … critical section … ; s.release
```


Concurrent Observers

```
class Observer
    attr sub: Subject
    initialization (s: Subject)
        begin sub := s ; s.attach (this) end
    method update …
end

class Subject
    attr a, n : set of Observer
    initialization a, n := {}, {}
    method attach (o: Observer)
        a := a ∪ {o}
    method notifyAll
        n := a
    action notifyOne
        when n ≠ {} do
            var o: Observers ;
            begin o :∈ n ; n := n – {o} ; o.update end
end
```

Call may block and and action notifyOne can be initiated again (or one of the methods be called)

Summary of Methodology

- **Language extensions:**
  - **No construct for threads!**
  - **Classes: attributes, methods, actions**
  - **Guards for synchronization:**

```
when b do S = await b ; S
```

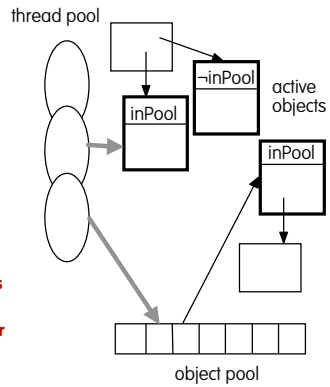
- **Implementation:**
  - **Automatic creation & management of threads; cf garbage collection!**
  - **Requires guards only over local attributes.**

- **Theory:**
  - **Formal model through action systems in higher order logic**
  - **Class verification & refinement: data refinement, atomicity refinement**

- **Goal: Bring the practice of concurrent object-oriented programming as close as possible to a simple model with a sound theory.**

Implementation

- active objects created in object pool
- thread obtains object from pool and executes enabled action or removes it from pool
- execution of method or action may place object in pool again
- object locked as long as method or action is executing
- fairness among actions and objects
- garbage collection unaffected!



→ object pool management takes constant time

→ constant memory overhead per active object

→ evaluation of guards only once when a thread is available due to local guards (cf. exponential back-off protocol)

Classification

- Following [Briot et al 98]:
  - serial (only one activity at any time) vs quasi-concurrent (several activities, but only one can progress) vs fully concurrent (several activities can progress simultaneously)
  - autonomous vs reactive objects (Java)
  - guards (Eiffel) vs body (Ada) for acceptance of calls
  - synchronous method calls vs message queues (Actors)

Action Systems (Concurrent Modules)

- **module** K ≙ K = (init, proc, act)

  var p : P := p_0
  var q : Q := q_0
  procedure m = M
  action a = A
  action b = B
  end

  init = (p_0, q_0)
  proc = M
  act = A ⊓ B

- Programs are composed of modules. Composing modules
  - combines their variables
  - combines their procedures
  - takes the nondeterministic choice of their actions: interleaving semantics.

  Module composition models both inheritance and usage.

- Module Refinement K $\sqsubseteq_R$ K' with relation R.

Classes, Objects, Attribute Selection, Method Call

- **class** C ≙ **module** C

  attr p: P
  initialization I
  method l = L
  method m = M
  action a = A
  end

  var C: **set of** Object := {}
  var p: Object ⟦ P
  procedure new: Object
  result :⟦ C ⟦ {nil} ; C := C ⟦ {result} ; I
  procedure l (this: Object)
  {this ⟦ C} ; L
  procedure m (this: Object)
  {this ⟦ C} ; M
  action a
  var this :⟦ C ; A
  end

Assuming c: Object:

- c.p ≙ C.p (c) **attribute selection**
- c := **new** C ≙ c := C.new **object creation**
- c.m ≙ C.m (c) **method call – without dynamic binding**

Delayed Doubler Refinement

```
class D
    attr x: integer
    method store (u: integer)
        x := 2 ⋅ u
    method retrieve: integer
        result := x
end
class DD
inherits D
    attr d: boolean
    initialization d := true
    method store (u: integer)
        x, d := u, false
    method retrieve: integer
        when d do
            result := x
    action double
        when ¬d do
            x, d := 2 ⋅ x, true
end
```

R (D, x) (DD, x', d) =
(D = DD) ∧
(∀ o ∈ D · (o.d ⇒ o.x' = o.x)
(¬o.d ⇒ 2 ⋅ o.x' = o.x))

Conditions for D $\sqsubseteq_R$ DD**:**

1. **for module initialization:**
   R ({}, x) ({},x', d)
2. **for method** store **(**new, retrieve **similarly):**
   a) **refinement:**
      D.store $\sqsubseteq_R$ DD.store
   b) **enabledness:**
      grd D.store ∨ R (D, x) (DD, y, d) ⇒
      grd DD.store grd double
3. **for the action** double:
   a) **refinement:**
      skip $\sqsubseteq_R$ DD.double
   b) **termination:**
      R (…) (…) ⇒ trm (**do** double **od**)

Delayed Doubler Invariant

```
class D
    attr x: integer
    method store (u: integer)
        x := 2 ⋅ u
    method retrieve: integer
        result := x
end
class DD
inherits D
    attr d: boolean
    initialization d := true
    method store (u: integer)
        x, d := u, false
    method retrieve: integer
        when d do
            result := x
    action double
        when ¬d do
            x, d := 2 ⋅ x, true
end
```

P = (∀ o ∈ DD · o.d ⇒ even(o.x))

Conditions for P **being an invariant of** DD**:**

1. **for module initialization:**
   (DD = {}) ⇒ P
2. **for the procedures:**
   {P} DD.new {P}
   {P} DD.store {P}
   {P} DD.retrieve {P}
3. **for the actions:**
   {P} DD.retrieve {P}

Summary and Outlook

- **Prototypical implementation with code generation for JVM**
- **Theory:**
  - **model based on the simple theory of types; all proofs are done in higher order logic**
  - **model simplified by being less restrictive than the compiler**
- **Related Theories:**
  - **OO action systems [Bosangue et al 98, 99]: atomicity of actions**
  - **Seuss [Misra 02]: atomicity of actions, pre-procedures**
  - **□□□□ [Jones 92, 96]: methodology, examples, early return**
- **Ongoing work:**
  - **extension of model with fairness**
  - **inclusion of exception handling**
  - **inclusion of specification constructs**
  - **separation of subtyping from subclassing**
  - **improved code generation for JVM**