A Model for Concurrent Object-Oriented Programming

Emil Sekerinski
McMaster University
Hamilton, Ontario, Canada
March 2004

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
  in, out, n, max, b := 0, 0, 0, m, new Object[m]
  method put (x: Object)
    when n < max do
      in, blin, 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 p into q:
  x := p.get(); if f |x| then q.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

Delayed Doubler Correctness

class DD
  attr x: integer
  attr d: boolean
  initialization d := true
  method store (u: integer) x := 2 x u
  method retrieve: integer
    result := x
end

class D
  attr x: integer
  attr d: integer
  initialization d := DD
  method store (u: integer) x := DD x u
  method retrieve: integer
    result := x
end

Conditions for D \subseteq DD:
1. for program initialization:
   R (\emptyset, x \emptyset) x, d

2. for method store (new, retrieve similarly):
   a) refinement:
      D.store \subseteq DD.store
   b) enabledness:
      grid DD.store \land \forall \exists DD, y, d \Rightarrow
grid DD.store \land grid double

3. for the action double:
   a) refinement:
      skip \subseteq DD double
   b) termination:
      R (...) (...) \Rightarrow trm \{ do double od \}

Introducing Concurrency

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

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
    when ¬d do
      x, d := 2 x x, true
end

• Typical pattern for file operations, network transmission, web browsers, ...

• Objects of class DelayedDoubler can be used wherever objects of class Doubler are expected. DelayedDoubler refines Doubler:

  Doubler \subseteq DelayedDoubler

Introducing Concurrency in Subclasses

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

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

Doubler

DelayedDoubler
Concurrent Priority Queue ...

```plaintext
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
end
```

```plaintext
Concurrent Leaf-oriented Trees

t := new Tree ;
t.add[3] ;
t.add[7] ;
t.add[2] ;
t.add[8] ;
t.add[1] ;
```

- Correctness:
  
  PriorityQueueSpecification ⊆ PriorityQueue

  - 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

**Class** Philosopher

- attr `state`: thinking, hungry, eating, full
- attr `left`, `right`: Fork
- initialization `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

Fairness through Strong Semaphore

**Class** WeakBinarySemaphore

- attr `a`: boolean
- initialization `a` := true
- method `acquire`
  - when `a` do
    - `a` := false
- method `release`
  - `a` := true

**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`)
  - end
- method `release`
  - `a` := true

- **Extra**
  - 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:

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

... Dining Philosophers

**Main program:**

```java
var fork := new Fork(5);
var philosopher := new Philosopher(5);

for i := 0 to 4 do
  fork[i] := new Fork();
for i := 0 to 4 do
  philosopher[i] := new Philosopher(fork[i], fork[(i + 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.**

Concurrent Observers

**Class** Observer

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

**Class** Subject

- attr `a`, `n`: set of Observer
- initialization `a`, `n` := {}, {}
- method `attach` (o: Observer)
  - `a` := `a` ∪ {o}
- method `notifyAll`

```java
n := a
action notifyOne
  when n ≠ {} do
    var a: Observers;
    begin a ∈ n; n := n − {a}; a.update end
  end
```

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

Classes, Objects, Attribute Selection, Method Call

- class C ∈ module C
  - attr p : P
  - initialization I
  - method m = M
  - action α = A
  - procedure new : Object
    - result : C U (nil) ; C := C U (result) ; I
    - procedure m (this : Object)
      - (this ∈ C) ; M
      - action α
      - var this : C ; A
  - end

For c : Object:

- c.p ∈ C.p (c) attribute access
- c := new C ∈ c := C.new object creation
- c.m ∈ C.m (c) method call

Summary of Language

- Language features:
  - No construct for threads.
  - Classes: attributes, methods, actions
  - Guards in methods and actions for synchronization:
    when b do S = await b ; S

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

Inheritance (simplified), Subtyping, Super-call, Type Test, Type Cast

- class C ∈ module C
  - attr p : P
  - method m = M
  - end

- module D import C
  - var D : set of Object := {} ; q : Object → Q
  - procedure new : Object
    - result : D U (nil) ; C := C U (result) ; I
    - procedure n (this : Object)
      - (this ∈ D) ; N
  - end

D is subtype of C means D ⊆ C. For c, d : Object:

- super.m ∈ C.m super-call in D
- c is D ∈ C ∈ D type test
- d := c as D ∈ (c ∈ D) ; d := c type cast
Action Systems (Concurrent Modules)

- module K  ⊆  K = (init, proc, act)
  var p : P := p0
  var q : Q := q0
  procedure m = M
  action a = A
  action b = B
end

- Module composition models both inheritance and usage:

  K0 = (initp, procp, actp)
  K1 = (initp, proc0, act1)
  K0 || K1 = (initp, init0, (proc0, proc1), actp ∩ act1)

Nondeterministic choice of actions: interleaving semantics.

- Class refinement reduced to module refinement K ⊆ R K’ with relation R

Related Work

- 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)

- Related Theories:
  - OO action systems [Bosangue et al 98, 99]: atomicity of actions
  - Seuss [Misra 02]: atomicity of actions, pre-procedures
  - πβλ, [Jones 92, 96]: methodology, examples, early return

Outlook

- Further work:
  - Refinement theory [Büchi & S 00, S 03]
  - Relaxed balanced AVL trees [Yan 04]
  - Prototypical Compiler for JVM [Lou 04, Yan 04]

- Ongoing theoretical work:
  - embedding in higher order logic
  - inclusion of fairness
  - inclusion of exceptions

- Ongoing implementation work:
  - inclusion of exception handling
  - specification constructs
  - separation of subtyping from subclassing
  - improved code generation for JVM
  - code generation for Mach kernel