Concurrent Object-Oriented Programming

Arnaud Bailly, Bertrand Meyer and Volkan Arslan

Coordination

- Objects have a **state**.
- The set of available routines may depend on the current state.
- The Queue exhibits such a **non-uniform** service.
- Activity-centered coordination:
  - clients coordinate to prevent interfering actions.
  - Invocations are never delayed.
- Objects are **input-enabled**.
- Boundary coordination:
  - the supplier may delay the acceptance of a call.
  - Introduction of **synchronization code**.

Issues about Synchronization code

- How to write it?
- Where to place it?
- Can it be re-used?

Activity-Centered Coordination

- Multiple Clients to 1 supplier:
  - the goal is **avoiding interference**!
  - 1 client accessing the supplier at a time.
  - This is **mutual exclusion**.
- One can use "traditional" mechanisms:
  - polling on a shared variable (busy-wait),
  - using semaphores,
  - using conditional critical regions, and
  - exchanging messages among clients.
- The **synchronization code is on the client side**.

Semaphores [Dijkstra 68]

- A semaphore s has:
  - an integer value val,
  - two **atomic** operations wait and signal,
  - a queue of waiting calls.
- When calling s.wait:
  - if val>0 then do val := val - 1 and return,
  - else suspend and queue this call.
- When calling s.signal:
  - if there is a queued call resume it,
  - else do val := val + 1.
Semaphore Implemented in Java

```java
class Semaphore {
    private int val;
    public Semaphore (int n) { val = n; }

    public synchronized sem_wait () { // P
        if (val <= 0)
            this.wait();
        val = val - 1;
    }

    public synchronized sem_signal () { // V
        this.notify(); //no effect if no thread is waiting
        val = val + 1;
    }
}
```

Semaphore as a Mutex in Java

```java
class TOOLBOX {
    private int resource, number;
    private Semaphore semaphore;
    public Example (int n) { number = n; semNewSemaphore (1); }

    private void exe() { sem_wait(); 
        // critical section
        resource = resource + 1;
        sem_signal(); // end of critical section
    }

    private void exe2() {sem_wait(); 
        // critical section
        resource = resource - 1;
        sem_signal(); // end of critical section
    }

    public static void main () {
        new Example (1).exe();
        new Thread (new Runnable (public void run () {exe2()});)
    }
}
```

Conditional Critical Region

- [Brinch Hansen 72] and [Hoare 72].
- Group variables into `resources`:
  - `resource v w ... z`;
- The `critical region` instruction is:
  - `region r when Condition
  then StatementList end`
- Executes StatementList only when:
  - Resources are available,
  - Condition is verified.
- Resources are held during execution and freed on termination.

Using CCR in SCOOP-Like Syntax

```java
class TOOLBOX // producer-consumer variation
create make
feature make (a,b, a c ; STREAM) is do a; add b; c = c end
merge is
do from
until false
loop
do_merge (a, b, c)
end
end
end
```

Using Message-passing

- Solving variants of the consensus problem.
- Implementing distributed mutual exclusion.
- Conserve the DSM exercise.
- Many other solutions:
  - distributed semaphores,
  - solutions tolerating faults...

Boundary Coordination

- The synchronization code is in the supplier class.
- Possible solutions:
  - live routines (POOL, ABCL/1),
  - monitors (Ada95),
  - delay Queues (Hybrid [Nierstrasz 92]),
  - behavior abstractions (ACT++ [Kafura & Lee 89]),
  - Enable Sets (Rosette [Tomlinson & Singh 89]),
  - method guards (Guide [Hagimont & al. 94]),
  - path expressions (Proc0l [Bos & Laffra 90])
- Along the way, find solutions to control intra-object concurrency.
Monitors

- A monitor is a language unit.
- It exhibits entries that can be called from outside.
- Only one entry executes at any time.
- Operations to interrupt and resume execution.
- Not unlike Java monitors.
- This allows clients to synchronize.
- All the synchronization code is enclosed inside the monitor.
- (A form of) modularity is obtained.
- Idea: chosen classes could yield monitor objects!
- Protected objects in Ada 95.

Semaphore in Ada 95

```ada
package Semaphore is
    protected type Semaphore (Initial : Natural := 0) is
        entry Wait; -- also known as P
        procedure Signal; -- also known as V
    private
        Value : Natural := Initial;
    end Semaphore;

    protected body Semaphore is
        entry Wait when Value > 0 is begin
            Value := Value - 1;
            end Wait;
        procedure Signal is begin
            Value := Value + 1;
            end Signal;
    end Semaphore;
```

Differences with Java

- Java uses condition variables.
- Ada uses conditional wait (no notify()).
- Java allows non-synchronized methods.
- Ada enforces synchronization among all entries.
- Java has one waiting queue per object.
- Ada has one waiting queue per entry.
- Java's are queues are unordered.
- Ada queues are FIFO.
- In Java, which object is notified is unknown.
- In Ada, it is the head of the queue.
- In Java, re-entrant calls are allowed.

The Eggshell Model

- In Ada, the notified task gets to be executed immediately (immediate resumption).
- Tasks inside the eggshell have priority over the ones outside.
- There is a requeue statement (even between distinct objects).

Delay Queues

- Each Object executes one routine at a time.
- Explicit management of queues, with queue objects.
- Each routine (entry) is linked to a queue.
- A queue Q can be either closed or open.
- There are methods Q.open() and Q.close().
- Very similar to include/exclude primitives.

BBuffer Using Hybrid Primitives

```ada
class BUFFER is
    public interface:
        put : (T; OBJECT); OBJECT get();
    implementation:
        private putQ, getQ : DELAY/QUEUE;
        Boolean isFull, isEmpty;
        put (x; OBJECT) linkPutQ x;
        getQ.read ();
        if (isEmpty) then putQ.close();
    end:
        OBJECT; get () link getQ is...
        putQ, read ()
        if (isEmpty) then getQ.close();
    end:
    end BUFFER;
```
Behavior Abstractions

- Classes have a behavior section.
- Behavior associates enabled routines to states.
- Become gives transitions between states.
- The state description is close to the interface.
- Related to Actor languages.

Enable Sets

- Enable sets are first-class objects.
- An enable set is a set of (allowed) method names.
- The both method allows combinations of sets.

```
class BUFFER is
  public interface: ...
  public implementation:
    empty: return enable(put);
    partial: return both(full(),empty());
    full: return enable(get);
    boolean isFull, isEmpty;

  public:
    BUFFER();
end BUFFER;
```

Method Guards (Guide)

- Each method has a guard that enables/disables it.
- Guards define (implicitly) the possible states.
- After each method execution guards are evaluated.
- There is no explicit transitions in the code!

```
class BUFFER is
  public interface: ...
  public implementation:
    put: if(full()) get: ifEmpty()
end BUFFER;
```

Path Expressions (Procol)

- A path expression is a regular expression.
- It defines a finite-state automaton.
- The disjunction represents non-determinism.

```
class BUFFER is
  public interface: ...
  public implementation:
    put: if(OBJECT) get: ifEmpty()
end BUFFER;
```
Controlling intra-object concurrency

- In GUIDE, for a method m the guards can feature:
  - invoked (m), started (m), completed (m),
  - current (m) = started (m) ≠ completed (m)
  - pending (m) = invoked (m) ≠ started (m)
- **Compatibility** annotations [Lühr 92] tag methods that can be executed in parallel with others.
- In C# (null): no annotation implies mutual exclusion,
  - "foo is --|-- bar-- --" foo and bar can execute in parallel,
  - "foo is --|-- ..." no restriction in parallelism.

Achieving Modularity [OOSC, Chap 3]

- A module is a programming artifact that is:
  - autonomous,
  - self-contained.
- In O-O languages: a module is a **class**.
- Respects the **Linguistic Modular Unit** and **Open-Closed Principles**.
- Respects the **information hiding** rule.
- Relevance can be assessed with criteria:
  - compositability and decomposability,
  - understandability,
  - continuity.
- If successful, obtain **components**.

Boundary Coordination

- Tends to break encapsulation:
  - clients aware of supplier’s state,
  - supplier implementation details do matter.
- Tends to make re-use harder.

```
class Client implements Thread {
    private Buffer b;
    public void run() {
        synchronized (b) {
            b.put(new Object());
            b.wake();
        }
    }
}
class Buffer {
    public synchronized notFull, notEmpty;
    public void put(Object o) {
        notFull.wait();
        put(o);
        notEmpty.notify();
    }
    public Object get() {
        notEmpty.wait();
        return get();
    }
}
```

Activity-centered coordination

- Makes supplier behavior **Encapsulated**.
- Convenience of re-use may vary.
- Synchronization code can be:
  - **interwoven** with functional code, or
  - **isolated** from it.
- Synchronization code should be **Separable**.

To be continued...

- Inheritance anomaly
- Contracts for concurrent objects