Active Objects

INF4140 - Models of concurrency
Active Objects, lecture 12

Høsten 2013

18.11.2013

Aims for this lecture

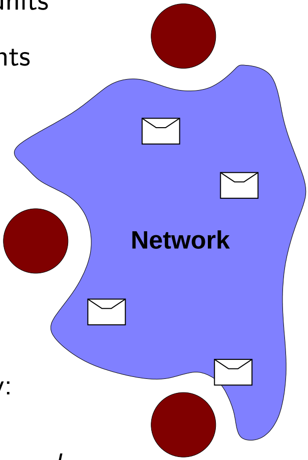
About distributed object-oriented systems and introduction to Creol

- Consider the combination of OO, concurrency, and distribution
- Understanding active objects
  - interacting by *asynchronous method calls*
- A short introduction into (a variant of) *Creol* using small example programs

Note: Inheritance and dynamic object creation not considered here.

Open Distributed Systems

- Consider systems of communicating software units
- **Distribution**: geographically spread components
  - Networks may be *asynchronous* and *unstable*
  - Component availability may vary over time
- **Openness** : encapsulation
  - Implementation of other objects is not necessary known.
  - Interaction with other objects is through interfaces.
- ODS *dominate* critical infrastructure in society: bank systems, air traffic control, etc.
- ODS: *complex*, *error prone*, and robustness is *poorly understood*



Challenges with OO languages for modern systems

Modern systems are often large and complex, with distributed, autonomous units connected through different kinds of networks.

- OO + concurrency
  synchronization, blocking, deadlock
- OO + asynchronous communication
  messages on top of OO or method-based communication?
  problems with RPC/RMI
- OO + distribution
  efficient interaction (passive/active waiting),
- OO + openness
  restricted knowledge of other objects
- OO + scalability
  management of large systems

Passive objects

- Execute their methods in the caller's thread of control (e.g., Java)
- In multithreaded applications, must take care of synchronization
  - Shared variable interference for non-synchronized methods
- If two objects call the same object, race condition may occur

Active (or concurrent) objects

- Execute their methods in their own thread of control (e.g., Actors)
- Communication is asynchronous
- Call and return are decoupled (future variables)
- Cooperative multitasking, specified using schedulers

Creol: A Concurrent Object Model

- OO modeling language that targets open distributed systems
- All objects are **active** (or concurrent), but may receive requests
  - Need easy way to combine active and passive/reactive behavior
- We don't always know how objects are implemented
  - Separate specification (interface) from implementation (class)
  - Object variables are **typed by interface**, not by class
- **No assumptions** about the (network) environment
  - Communication may be unordered
  - Communication may be delayed
  - Execution should adapt to possible delays in the environment
- Synchronization decided by the caller
  - **Method invocations** may be *synchronous* or *asynchronous*

- Object variables (pointers) are *typed by interfaces*
  (other variables are typed by data types)
- *Mutual dependency*: An interface may require a *cointerface*
  - Only objects of cointerface type may call declared methods
  - Explicit keyword *caller* (identity of calling object)
  - Supports callbacks to the caller through the cointerface
- All object interaction is *controlled* by interfaces
  - *No explicit hiding* needed at the class level
  - Interfaces provide behavioral specifications
  - A class may implement a number of interfaces
- **Type safety**: no "method not understood" errors

Interfaces

- Declares a set of method signatures
- With *cointerface* requirement

$$\textbf{interface} \;\; I \;\; \textbf{inherits} \;\; \overline{I} \;\; \textbf{begin}$$
$$\quad \textbf{with} \;\; J \;\; \overline{MtdSig} \;\; // \text{ cointerface } J$$
$$\textbf{end}$$

- Method signatures ($MtdSig$) of the form:

$$\textbf{op} \;\; m \;\; (\textbf{in} \;\; \overline{x : I} \;\; \textbf{out} \;\; \overline{y : I})$$

  - method name m with in-parameters $\overline{x}$ and out-parameters $\overline{y}$
  - Parameter types may also range over data types ($Bool$, Int, $String \ldots$)

- Consider the mini bank example from last week
- We have `Client`, `MiniBank`, and `CentralBank` objects
- Clients may support the following interface:

```
interface Client begin
  with MiniBank
    op pin(out p : Int)
    op amount(out a : Int)
end
```

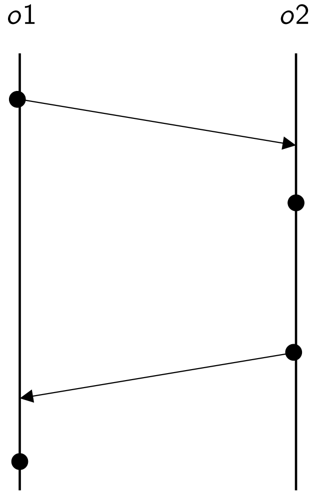
- only `MiniBank` objects may call the `pin` and `amount` methods

MiniBank and CentralBank interfaces:

```
interface MiniBank begin
  with Client
    op withdraw(in name : String out result : Bool)
end

interface CentralBank begin
  with MiniBank
    op request(in name : String, pin : Int, amount : Int
               out result : Bool)
end
```

Asynchronous Communication Model



- Object $o1$ calls some method on object $o2$
- In $o2$: Arbitrary delay after invocation arrival and method startup
- In $o1$: Arbitrary delay after completion arrival and reading the return

Main ideas of Creol: Programming perspective

Main ideas:

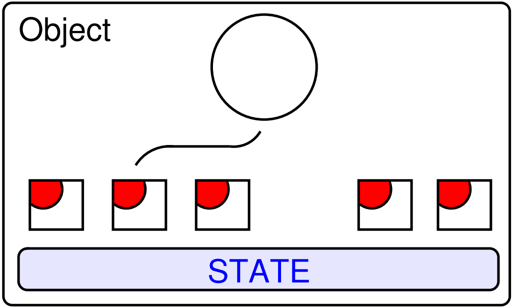
- Asynchronous communication
- Avoid undesired inactivity
  - Other processes may execute while some process waits for a reply
- Combine active and reactive behavior

In the language, this is achieved by statements for

- asynchronous method calls and
- processor release points

Note: Relase points enable *interleaving* of *active* and *reactive* code
Note: No need for signaling / notification

- **Concurrent objects** encapsulate a processor
- Execution in objects should *adapt* to environment delays
- At most *one active process* at a time
- *Implicit scheduling* between internal processes inside an object



Internal Processes in Concurrent Objects

- **Process** (method activation): code + local variable bindings (local state)
- **Object**: *state* + *active* process + *suspended* processes
- **Asynchronous** invocation: $t!o.m(In)$
  - The *label* t identifies the call

  Reading the result: $t?(Out)$
- **Processor release points**
  - Declared by **await** statements: **await** *guard*
  - *Guards* can be
    - $t?$
    - Boolean condition
    - and also method call
  - If a *guard* evaluates to false the active process is *suspended*
  - If no process is active, any suspended process may be *activated* if its guard evaluates to true.

- Objects communicate through method invocations *only*
- *Different ways to invoke* a method m
- Decided by caller — *not* at method declaration site
- **Guarded** invocation:

$$t!o.m(In); \ldots; \textbf{await } t?; t?(Out)$$

- **Label free abbreviations** for standard patterns:
  - $o.m(In; Out) = t!o.m(In); t?(Out)$ — **synchronous call**
  - $\textbf{await } o.m(In; Out) = t!o.m(In); \textbf{await } t?; t?(Out)$
  - $!o.m(In)$ — no reply needed
- **Internal calls:** $m(In; Out)$, $t!m(In)$, $!m(In)$
  Internal calls may also be asynchronous/guarded

Syntactic categories.	*Definitions.*
t in Label	$g ::= \phi \mid t? \mid g_1 \wedge g_2$
g in Guard	$p ::= o.m \mid m$
p in MtdCall	$\mathsf{S} ::= s \mid s; \mathsf{S}$
S in ComList	$s ::= \textbf{skip} \mid \textbf{begin } \mathsf{S} \textbf{ end} \mid \mathsf{S}_1 \Box \mathsf{S}_2$
s in Com	$\mid x := e \mid x := \textbf{new } \mathit{classname}(e)$
x in VarList	$\mid \textbf{if } b \textbf{ then } \mathsf{S}_1 \textbf{ else } \mathsf{S}_2 \textbf{ end}$
e in ExprList	$\mid \textbf{while } b \textbf{ do } \mathsf{S} \textbf{ end}$
m in Mtd	$\mid\ !p(e) \mid t!p(e) \mid t?(x) \mid p(e; x)$
o in ObjExpr	$\mid \textbf{await } g \mid \textbf{await } p(e; x)$
b in BoolExpr	$\mid \textbf{release}$

- Omit the functional language for expressions **e** here: this, caller, strings, integers, lists, sets, maps, etc

```
class Bank implements CentralBank begin
  var pin  -- pin codes, indexed by name
  var bal  -- balances, indexed by name

  with MiniBank
    op request(in name : String, pin : Int, amount : Int
               out result : Bool) ==
      if (pin[name] = pin && bal[name] >= amount)
        then result := true
        else result := false end
end
```

```
class MiniBank(bank : CentralBank) implements MiniBank begin
  with Client
    op withdraw(in name : String out result : Bool) ==
      var amount : Int, pin : Int;
      caller.pin(;pin); caller.amount(;amount)
      await bank.request(name, pin, amount; result)
end
```

- method calls `caller.pin(...)` and `caller.amount(...)` are type safe by cointerface requirements
- **await** statement: passive waiting for reply from `CentralBank`

Example: Client implementation

Optimistic client:

```
class Person(m : MiniBank) implements Client begin
  var name : String, pin : Int;

  op run == success : Bool;
    await m.withdraw(name;success);
    if (success == false) then !run end

  with MiniBank
    op pin(out p : Int) == p := pin
    op amount(out a : Int) == a := 1000
end
```

- Assuming communication with a fixed minibank m

Main ideas of Creol: Programming perspective

- concurrent objects (each with its own virtual processor)
- a notion of asynchronous methods calls, avoids blocking, using processor release points
- high level process control
  - no explicit signaling/notification
  - busy waiting avoided!
- openness by a notion of multiple interfacing
- type safe call-backs due to cointerfaces

  Remark: abstraction by behavioral interfaces

Example: Buffer

```
interface Buffer begin
  with Producer op put(in x : Int)
  with Consumer op get(out x : Int)
end

class OneSlotBuffer implements Buffer begin
  var value : Int, full : Bool;
  op init == full := false
  with Producer
    op put(in x : Int) == await ¬full; value := x; full := true
  with Consumer
    op get(out x : Int) == await full; x := value; full := false
end
```

- init: initialization code executed at object creation

Example: Buffer (cont.)

Illustrating alternation between active and reactive behavior

```
class Consumer(buf: Buffer) implements Consumer begin
  var sum : Int := 0;
  op run == var j : Int;
    while true do await buf.get(;j); sum := sum + j end
  with Any op getSum(out s : Int) == s := sum
end
```

- Call to `buf.get`:
    - Asynchronous
    - **await**: processor release
    - Incoming calls to `getSum` can be served while waiting for reply from `buf`
- Interface `Any`: supertype of all interfaces
    - Any object can call `getSum`

Readers/Writers example (Simple implementation)

```
interface RW
begin with RWClient
    op OR — open read
    op OW — open write
    op CR — close read
    op CW — close write
end

class RW implements RW
begin var r: Int:=0; var w: Int:=0;
with RWClient
    op OR == await w=0; r:= r+1
    op OW == await w=0 and r=0; w:= w+1
    op CR == r:= r-1
    op CW == w:= w-1
end
```

Note: A client may do asynchronous calls to OR/OW and synchronous calls to CR/CW.

Readers/Writers example (version 2)

```
class RW(db : DataBase) implements RW begin
  var readers : Set[Reader] := ∅, writer : Writer := null,
    pr : Int := 0; // number of pending calls to db.read

  with Reader
    op OR == await writer = null; readers := readers ∪ caller
    op CR == readers := readers \ caller
    op read(in key : Int out result : Int) ==
      await caller ∈ readers;
      pr := pr + 1; await db.read(key;result); pr := pr - 1;

  with Writer
    op OW == await (writer = null && readers = ∅ && pr = 0);
      writer := caller
    op CW == await caller = writer; writer := null
    op write(in key : Int, value : Int) ==
      await caller = writer; db.write(key,value);
end
```

RW example, remarks (version 2)

- `read` and `write` operations on database may be declared with cointerface `RW`
- Weaker assumptions about `Reader` and `Writer` behavior than in the first version
  - Here we actually check that only registered readers/writers do read/write operations on the database
- The database is assumed to store integer values indexed by `key`
- Counting the number of pending calls to db.read (variable `pr`)
- A reader may call `CR` before all `read` invocations are completed
- For writing activity, we know that there are no pending calls to `db.write` when `writer` is **null**. Why?
- The solution is unfair: writers may starve
- Still, after completing `OW`, we assume that writers will eventually call `CW`. Correspondingly for readers

Summary: Active Objects

- Passive objects usually execute their methods in the thread of control of the caller (Java)
- In multithreaded applications, we must take care of proper synchronisation
- Active objects execute their methods in their own thread of control
- Communication is asynchronous
- synchronous communication possible by means of asynchronous communication primitives
- Call and return are decoupled by the use of *labels*
- Usually, active objects use *cooperative* multitasking.
- Cooperative multitasking is specified using *schedulers*. Our scheduler will just randomly pick a next process.

PMA Group Courses

Spring:

- INF3230 - Formal modeling and analysis of communicating systems
  rewriting logic - language and tool Maude
- INF5140/INF9140 - Specification and verification of parallel systems. ('11, '13, '15, ...)
  Automatic verification using model checking techniques
- INF5906/INF9906 - Selected topics in static analysis. ('10, '12, '14, ...)
  analysis of programs at compile time

Fall:

- INF5130/INF9130 - Selected topics in rewriting logic ('11, '13, '15, ...)

Each semester:

- INF5160 - Seminar in Computer Science ("Formal methods seminar")