CONCURRENT OOP WITH AGENTS
ALEOO LANGUAGE

ALESSANDRO RICCI, ANDREA SANTI
University of Bologna, Italy

AGERE! @ SPLASH 2013 - Indianapolis, Indiana (US) - Oct 28, 2013
MOTIVATIONS

simpAL (AGERE! 2011)

- agent-oriented programming as a programming paradigm
- complex model
  - agents, artifacts, tasks, objects, organizations...

NEXT STEP

- finding a simpler model yet preserving the key features of the agent-oriented abstractions
- stronger integration with objects
ACTIVE + PASSIVE

• Well-known and discussed issue in literature
  ‣ combining
    \{processes/actors/active objects/... \}
    and objects

• Relevant issue today in the practice
  ‣ e.g. actor libraries + passive objects

• Beyond mechanisms & tricks
  ‣ looking for a sound integration from a conceptual and design point of view
EXISTING PROPOSALS

• active objects as *extensions of objects*
  ▸ 1-1 mapping + further features
  ▸ ex: Hybrid, Caromel’s Async Distrib Objects, Creol, Eiffel//,...

• actors as *containers of objects*
  ▸ actors are *not* objects and are not at the same level of abstraction of objs
  ▸ ex: VAT model, CoBoxes, Cogs
ACTORS AS CONTAINERS

- Objects encapsulated inside actors
  - sharing the event loop
- Asynchronous + synchronous interaction among objs
  - can be synchronous only if they are owned by the same actor
- Example: VAT (E, AmbientTalk)

---

This enqueues the requests to withdraw and deposit in the main actor's own message queue. However, the programmer should now be aware that other messages may happen to arrive after withdraw but before deposit was scheduled. In other words, the transfer is no longer atomic, which may or may not be a problem, depending on application requirements.

5.2 Far References

It is possible for objects owned by one actor to hold references to individual objects owned by other actors. Such references that span different actors are named far references (the terminology stems from E [MTS05]) and only allow asynchronous access to the referenced object. This ensures by design that all communication between actors is asynchronous. Trying to perform a sequential method invocation on a far reference provokes a runtime exception.

---

Figure 1: AmbientTalk actors as communicating event loops. The dotted lines represent the event loop activity of each actor which perpetually takes the next message from its message queue and executes the corresponding method on one of its owned objects.

To illustrate far references, consider the following example. Our main actor spawns a new actor and decides to share its account objects with this new actor:

```def helper := actor: {
def transfer(from,to) {
  from<-withdraw(10);
  to<-deposit(10);
}
}
```

The expression `actor:`{...} spawns a new actor. The new actor immediately creates a new object, as if by evaluating `object:`{...}. This object, call it `o`, acts as the actor's public interface. The `actor:` expression immediately evaluates to a far reference to `o`, which is stored in the `helper` variable. The main actor then sends the `transfer` message via a far reference to `o`. Messages sent via a far reference to an object are enqueued in the message queue of the object's owner for later processing. Hence, the `transfer` message will later be dequeued and executed by the new actor, not by the main actor.
CONTRIBUTION

• Yet another approach about integrating active and passive entities...
  ‣ ...based on agent-oriented abstractions
  ‣ ...conceptually different from the existing ones
  • agents are neither an extension of objects, nor containers..

> ALOO model and language (simpAL-OO)
  ‣ Concurrent OOP with
    OOP + agent-oriented programming
ALOO Concepts
ALOO program (in exec) = organization of agents working in the same environment represented by a set of objects.

- **agents** (active)
  - autonomous entities in charge of tasks to do
  - encapsulating control

- **objects** (passive)
  - passive entities used & observed by agents
  - source of actions and observable state/events

- **organizations**
  - ensemble of agents and objects
  - structured in workspaces (localities)
AGENT(S)

OBJECT(S)

actions

obs prop
direct access

observable properties

observable events

State

Plan lib

Event queue

task(s) todo

plan in exec

...
AGENT-OBJECT INTERACTION: THE ACTION MODEL

• Actions
  ‣ created when an agent invokes an op
  ‣ can complete with success (& output param)
  ‣ can fail (with failure info)

• Atomic execution
  ‣ no races on obj state
  ‣ changes to obs props are perceived atomically (when the action complete/fail)
INTER-OBJECTS INTERACTION

- Actions spanning on multiple objects
  sequential *synchronous* behaviour

- to avoid deadlocks:
  - strong/weak atomicity if the invoked obj is either private (by construction) or not
  - if not private => invoking op is suspended, releasing the lock & committing a new obs state (weak atomicity)
AGENTS

ACTORS

AGENT EMBEDDED CONTROL LOOP

```plaintext
loop {
    ev = poll(evq)
    acts = select(state, plans_in_exec, ev)
    perform (acts, state)
}
```
AGENT PROGRAMMING MODEL

• roles
  ‣ agent type

• scripts
  ‣ agent implementation
• declaring what types of tasks an agent of that type (= playing that role) is capable to do

```plaintext
role FooRole {
  tasks:
    FooTaskA, FooTaskB
}

...
myAgent: FooRole
```
AGENT SCRIPTS

agent-script FooScript plays FooRole {

  x: int, c: Count /* global state */

  plan-for FooTaskA { ... }
  plan-for FooTaskB { ... }

}

• agent-level vars + plans definition
• at least one plan for each task declared in the role
• Plan body => *action rule block* \{ . . . \}

• Action rule blocks => unit of behavior describing *what* actions to do *when*
  
  ‣ set of *action rules*  
    \[ e \mid c \Rightarrow a \ [1] \]  
    
  ‣ + sugar for frequent patterns of actions

  ‣ local vars

  ‣ attributes

    ‣ objects to be observed, when completed, when to be repeated, interruptible/critic
CountingTask

n: int = 0

this-env.stdout.println(msg: "starting");

observing:
this-task.
tool as: c {
  #to-be-repeated
  c.inc();
  this-task.env.println(msg: "inc done")
}

every-time changed: c.count as: v => {
  this-env.stdout.println(msg: "perceived: "+v);
  n = n + 1
}

}
plan-for FooTask {
    s1, s2: Service res: int

    s1: Service <= new-obj ACMEServiceA();
    s2: Service <= new-obj ACMEServiceB();
    {
        #completed-when: is-done printRes

        s1.request(result: res) #act:r1
        s2.request(result: res) #act:r2

        when is-done r1 || is-done r2 => {
            this-env.stdout.println(msg: "result: "+res)
        } #act: printRes
    }
}
AGENT SCRIPTS

agent-script FooScript plays FooRole {

  private-tasks: SubTask1, SubTask2
  ...
  plan-for SubTask1  {  ...  }
  plan-for SubTask2  {  ...  }

}

• private tasks, not part of roles
  ▷ self-assigned
PLAN DECOMPOSITION

- Creating & self-assigning sub-tasks
  - **do-task** => using same plan stack (~proc. call)
  - **start-doing-task** => new plan stack

```
agent-script CalcAgent plays Calculator {
  plan-for FactTask {
    if (this-task.n <= 1){
      this-task.set(result: this-task.n)
    } else {
      t: FactTask <= new-obj FactTask(n: this-task.n - 1);
      do-task t;
      this-task.set(result: this-task.n*t.result)
    }
  }
}

interface FactTask {
  n: int
  result: int #out
}
```
• Operations provided by predefined built-in objects
  ‣ `new-agent` - Script + task to do

```agent-script
CalcAgent plays Calculator {
  plan-for FactTask {
    if (this-task.n == 1){
      this-task.set(result: this-task.n)
    } else {
      t: FactTask <= new-obj FactTask(n: this-task.n - 1);
      observing: t {
        #completed-when: is-done set
        new-agent CalcAgent() task: t
        when changed: t.result as: res =>
          this-task.set(result: this-task.n*res) #act: set
      }
    }
  }
}
```
agent-script DragAndDropScript plays GUIManagerRole {

  plan-for DragAndDropTask {
    canvas: Canvas <= new-obj testmix.canvas.CanvasObj();
    canvas.init(title: "Test Drag&Drop");
    observing: canvas {
      canvas.show()

      every-time changed: canvas.pressed as: pressed | pressed => {
        #completed-when: is-done drop

        this-env.stdout.println(msg: "dragging...")

        every-time changed: canvas.mousePos as: p => {
          this-env.stdout.println(msg: "x: "+p.x+" y: "+p.y);
          canvas.drawPoint(pos: p)
        }
        when changed: canvas.pressed as: stillPressed | !stillPressed => {
          this-env.stdout.println(msg: "drop")
        } #act: drop
      }
    }
  }
}
FURTHER ASPECTS

• task/plan decomposition
  ‣ creating and self-assigning tasks from a plan

• management of action failures
  ‣ failed events + is-failed predicate on action label

• critical action blocks
  ‣ to avoid interleaving of plan execution in the case of an agent executing multiple plans

• agent dynamic creation

• ...

OBJECTS AS
COORDINATION MEDIA

interface Barrier {
    nParticipants: int
    nHits: int
    sync()
}

class SimpleBarrier implements Barrier {
    SimpleBarrier(nParticipants: int){ nHits = 0; }

    sync(){
        nHits = nHits + 1;
        await (nHits >= nParticipants);
    }
}
ORG PROGRAMMING

- org-scripts
  - declaration of the initial set of agents and objs, partitioned in workspaces
  - possibly mapped onto different nodes
  - the “main” of an ALOO program is an org-script

```org-script
SimpleTest  {
    workspace main {
        t: Testing <= new-obj Testing()
        new-agent Tester() tasks: t
    }
}
```
ORGs AS COMPUTATIONAL ENTITIES

• orgs can be created at runtime, as sub-orgs of existing orgs
  ▸ first-class computational entities
  ▸ same type of agents (role)
    • created with a task to accomplish
• coarse-grained modularity
• hierarchical structuring
org-script SubOrg plays FooRole {
  org-config-for FooTask {
    workspace main {...}
    workspace alfa {...}
    ...
  }
}

/* in an agent plan */
...

t: FooTask <= new-obj FooTask(...);
o: FooRole <= new-org SubOrg() task:t
ALOO
IMPLEMENTATION & EVALUATION
CURRENT IMPLEMENTATION

• Java-based parser + ~compiler
  ‣ based on Eclipse Xtext
  ‣ binaries ~ set of Java objects fetched by the runtime

• Java-based runtime
  ‣ simple thread pool strategy with work stealing
  ‣ one for agents (cycles) + one for actions

• simple IDE
  ‣ based on Eclipse
MAIN CHALLENGES

• agent control loop efficiency
  ‣ cycling by need techniques

• action execution
  ‣ locking by need
  ‣ avoid thread switching when possible

• object instantiation
  ‣ use lightweight objects when possible
# FIRST PERFORMANCE EVALUATION

<table>
<thead>
<tr>
<th></th>
<th>cs</th>
<th>cm</th>
<th>cr</th>
<th>cg</th>
<th>pd</th>
<th>ac</th>
<th>es</th>
<th>em</th>
<th>tr</th>
</tr>
</thead>
<tbody>
<tr>
<td>AF</td>
<td>0,08</td>
<td>0,3</td>
<td>9,3</td>
<td>1,6</td>
<td>-</td>
<td>0,26</td>
<td>4,1</td>
<td>3,4</td>
<td>2</td>
</tr>
<tr>
<td>ALOO</td>
<td>13,6</td>
<td>7,5</td>
<td>14,6</td>
<td>2,5</td>
<td>0,13</td>
<td>0,12</td>
<td>12,3</td>
<td>4,5</td>
<td>35</td>
</tr>
<tr>
<td>Jason</td>
<td>333</td>
<td>238</td>
<td>334</td>
<td>38,7</td>
<td>0,33</td>
<td>3,1</td>
<td>144,2</td>
<td>37,1</td>
<td>41,4</td>
</tr>
</tbody>
</table>
NEXT STEPS

• formalization
  ‣ core language & operational semantics
• investigating important issues
  ‣ sub-typing (for roles in particular)
  ‣ inheritance (for agent-scripts)
• modeling & programming guidelines/method
• stressing the language with case studies