Concurrent Object-Oriented Programming with Agents

Alessandro Ricci
University of Bologna, Italy
a.ricci@unibo.it

Andrea Santi
University of Bologna, Italy
a.santi@unibo.it

Abstract

ALOO is a novel approach to Concurrent Object-Oriented Programming, integrating plain old objects with concurrency through the adoption of *agent-oriented* first-class abstractions.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features; D.1.3 [*Programming Techniques*]: Concurrent Programming

Keywords Concurrent Object-Oriented Programming; Agents

1. Introduction

The conceptual integration of OOP with concurrency [2] – including asynchronous and event-driven programming – is still an issue, both from a conceptual and practical point of view. Proposals in the state-of-the-art can be broadly classified in two opposite families. On one extreme we have those proposals that keep essentially the good old "mainstream" OOP with a support for multi-threading and add *ad hoc* abstractions and mechanisms to simplify multi-threaded and asynchronous programming. On the other extreme we have proposals that revise the foundations, injecting concurrency at the core of the object model. A main example is given by *actors* [1] and related approaches like concurrent/active objects [6]. Each family has some pros and cons and this leads developers to mix different concurrent models in practice, for instance integrating actors, passive objects, and threads [5]. This makes programming quite tricky and often leads to solutions with a poor design.

In this work we introduce a novel approach to Concurrent Object-Oriented Programming based on a simple conceptual model integrating plain old objects with *agents*, introduced as first-class abstraction for modeling the active parts of the program. The model is implemented in a language/platform called ALOO[1]. and is based on previous work done in the context of the simpAL project [3]—where agent-oriented abstractions were introduced but without being integrated with objects. The ALOO name is the short version of simpAL-OO.

A program (system) in ALOO is modelled as an *organization* of task-driven autonomous agents that work together inside a shared environment represented by a set of objects, that they cooperatively use, observe, create. The organization abstraction is used to explicitly define a collection of agents and objects, structured in *workspaces*. Workspaces are logical containers defining a notion of locality–in the case of organizations distributed over multiple network nodes. In the simple example shown in Figure 1, the program is given by a couple of Worker agents working inside a main workspace, doing cooperatively a Counting task, sharing a Count object. Classes implementing the interfaces are not reported.

The background metaphor underlying the approach is given by human organizations where people (agents) work cooperatively by exploiting resources and tools (objects), representing either instruments or the results of their job.

On the one side, the approach shares some key features of the actor model—decoupling of logical and physical concurrency, strong encapsulation (of control) and modularity, abstraction level avoiding locks and related low-level mechanisms, event-driven programming without inversion of control. On the other side, it allows for also exploiting features such as safe (without races) sharing and use of passive objects, synchronous communication and coordination without deadlocks due to wrong use of locks. Besides, the agent model natively extends actors reactivity with a form of *pro-activity*, which allows us to describe and more easily implement goal/task-driven behaviours.

2. Agents and Objects in ALOO

Differently from threads and analogously to actors, agents in ALOO are logical concurrent entities, so you can have thousands of agents running on top of a few number of physical threads/processes. Like actors, agents encapsulate a state, a behaviour and the *logical thread of control* of such a behavior. Differently from actors, agents are not only reactive but also *pro-active*, in the sense that they are spawned with some explicit *task* to-do (e.g. a CountingTask in the example) – that could be shared/cooperative, not only individual. As soon as they complete their task, they terminate—so no garbage collection is needed. The type of an agent is called *role* (e.g. Worker in the example), and allows to specify the list of tasks that agents that declare to play that role must be capable to accomplish.

The behaviour of an agent is governed by a control loop [4], repeatedly selecting and executing actions to accomplish its task(s), possibly reacting to relevant events perceived from the environment (the objects). The programmer encodes the practical knowledge to accomplish tasks into *plans*, collected in *scripts*—that are interpreted and executed by the agent. Plans are similar to procedures, with the important difference that they are not a simple sequence of statements but a collection of *action rules*, each one specifying when (condition, event) execute some action—that can be a further block of action rules. Inside a plan the this-task variable refers to the actual task (object) to accomplish, while this-env refers to the object storing information about the system environment (including e.g. standard output, referred by the stdout observable property).

[1] On ALOO web site (`http://aloo.sourceforge.net`) the interested readers can find a technical report providing details about the language, as well as a first prototype of the platform with simple examples.

SPLASH '13, October 26–31, 2013, Indianapolis, Indiana, USA.
ACM 978-1-4503-1995-9/13/10.
http://dx.doi.org/10.1145/2508075.2508095

```
interface Count {
  count: int
  inc()  }

interface CountingTask { tool: Count }

role Worker { tasks: CountingTask }

agent-script WorkerScript plays Worker {
  plan-for CountingTask {
    jobDone: boolean = false
    observing: this-task.tool as: t {
      #completed-when: jobDone
      /* a block repeating a sequence of actions */
      { #to-be-repeated
        this-env.stdout.println(msg: "..working.."); t.inc()}
      /* reactions */
      every-time changed: t.count as: v => {
        this-env.stdout.println(msg: "new value: "+v);
        if (v > 1000){ jobDone = true }}
    };
    this-env.stdout.println(msg: "job done.") }}

org-script SimpleExample {
  workspace main {
    c: Count <= new-obj CountImpl(count: 0)
    t: CountingTask <= new-obj CountingTaskImpl(tool: c)
    a1: Worker <= new-agent WorkerScript task: t
    a2: Worker <= new-agent WorkerScript task: t }}
```

Figure 1. A program example in ALOO.

Such a model has been devised so as to allow for effectively implementing behaviours composed by structured workflows mixing actions (sequence, parallel,..) and reactions to events [4], without incurring problems like inversion of control or asynchronous spaghetti. In the example, the plan for the CountingTask in WorkerScript accounts for repeatedly incrementing the counter, logging some msg on standard output (pro-active part); besides, every time a new value of the count observable property of the counter is perceived, the agent reacts by *suspending* what is doing (interrupt-like behaviour), in order to print a message and do some checks for eventually terminating the job. We believe that the capability of easily expressing behavior integrating pro-activity and re-activity is a key distinguishing feature of agents with respect to actors, active objects and related abstractions existing in literature.

Objects in ALOO are passive entities, analogously to plain old objects, with their own identity and independent existence from agents. They encapsulate set of actions (analogous to methods) that agents may invoke (e.g. inc() in the example) and a set of variables called *observable properties* (e.g. count, tool), both part of the object interface (i.e. type), representing object observable state that agents can perceive in an event-driven fashion. Besides, objects can have an inner, not observable state. Also tasks at runtime are uniformly represented by objects with a proper interface.

3. Use, Observation and Safe Sharing

Conceptually, agent-object interaction is *not* based on message passing (like in actors), neither on procedure call – but on action invocation (by agents on objects) and perception of asynchronous events generated by changes to objects' observable state (see Figure 2). Agents interaction is always mediated by objects—used and observed: direct communication among agents can be modeled on top, using suitably designed communication objects like message boxes, channels, blackboards and so on.

Actions executed inside an object can change its (observable or not) state and possibly invoke other actions over other objects. To make this interaction model effective, the object model has been devised so as to enforce safe sharing, so that multiple agents can work concurrently on the same shared objects without incurring

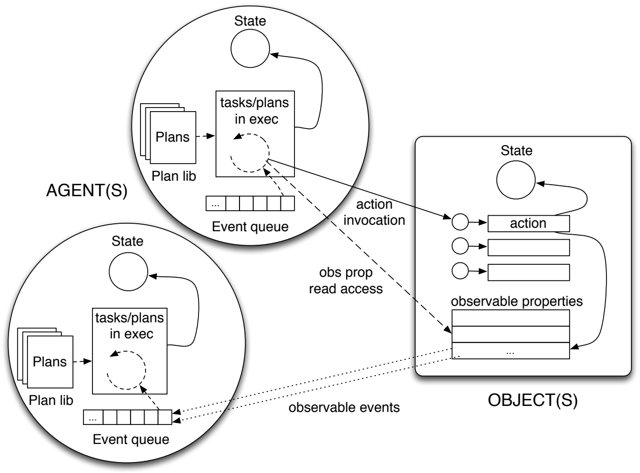

Figure 2. Agents and objects in ALOO

races. Similarly to monitors, inside an object, only one action can be in execution at a time and the effect of action execution on the object is made observable only when the action has completed. Differently from processes with monitors, the interaction model does not imply any (logical/physical) control coupling between agents and objects. Action completion/failure can be perceived by the agent as an asynchronous event, fetched by the agent control loop.

Besides actions, a main part of the interaction model is given by *observation*, which is natively supported by the model: an agent can dynamically decide to observe some group of objects (observing: construct) and, consequently, it automatically perceives an asynchronous event for every atomic change of the observable state of those objects, with guarantees about the ordering of events.

4. The Task Framework

Tasks are a key concept of the approach, founding agents' pro-activity and representing an important conceptual bridge to the design level. Besides the enabling mechanisms for task handling and management provided directly by the language, a Task Framework (TF) is provided on top of it, including a domain specific language to ease the description of complex tasks. The objective of the framework is to ease the definition of complex cooperative tasks, implementing a direct support for patterns of task organization and coordination which are recurrent in concurrent programming.

References

[1] G. Agha. Concurrent object-oriented programming. *Commun. ACM*, 33(9):125–141, Sept. 1990.

[2] J.-P. Briot, R. Guerraoui, and K.-P. Lohr. Concurrency and distribution in object-oriented programming. *ACM Comput. Surv.*, 30(3):291–329, 1998.

[3] A. Ricci and A. Santi. From actors to agent-oriented programming abstractions in simpAL. In *Proc. of SPLASH '12*, pages 73–74, New York, NY, USA, 2012. ACM.

[4] A. Ricci and A. Santi. Programming abstractions for integrating autonomous and reactive behaviors: an agent-oriented approach. In *Proc. of AGERE!'12*, pages 83–94, New York, NY, USA, 2012. ACM.

[5] S. Tasharofi, P. Dinges, and R. Johnson. Why do scala developers mix the actor model with other concurrency models? In *ECOOP'13, Montpellier, France, July 1–6, 2013*, 2013.

[6] A. Yonezawa, J.-P. Briot, and E. Shibayama. Object-oriented concurrent programming in ABCL/1. In N. K. Meyrowitz, editor, *OOPSLA*, pages 258–268. ACM, 1986.