From Actor Event-Loop to Agent Control-Loop – Impact on Programming

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

Abstract

Event-loops and control-loops are the main control architectures adopted respectively in actors and in agents. These architectures have a strong impact on the principles and discipline that can be adopted to design and program actors and agents. In this paper we develop this point, considering some main models/languages/technologies – ActorFoundry, Akka Actors, SALSA, AmbientTalk on the actor side and Jason and ALOO on the agent side – discussing and comparing them.

1. Introduction

The event-loop and the control-loop are two main control architectures adopted in literature to define the runtime behaviour of actors and agents. Actually, these architectures have been deployed in various computer science contexts, in different forms and complexity. A main example is given by operating systems, where event-loops have been largely used to define the control architecture of GUI-based applications. More recent examples include web applications – both at the client side, ruling the execution of JavaScript scripts, and at the server side, adopted by technologies such as Node.js – and mobile applications – adopted e.g. by the Android platform ruling the execution of activities. Control-loops have been adopted in Autonomic Computing [19] to define the behaviour of autonomic entities, based on the MAPE cycle- and, more generally, in self-adaptive system design [5]. Generally speaking, they are used to define the execution cycle of computing systems – being them full applications or individual components – that must be autonomous, from a control point of view, and must be capable to react to changes in their surrounding environment, and act accordingly, given some design objective.

With actors and agents, event-loops and control-loops are brought down to the computational model of the basic first-class abstractions adopted to design the active part of programs, so that a system or application is organized in terms of a possibly large number of active entities whose execution is loop-based. The properties of the computational model of e.g. actors based on event-loops have been already remarked in literature (e.g. in [21]). What we believe is missing is an analysis and discussion about how the adoption of such control architectures could impact on the programming model, and more generally on the programming principles and discipline promoted to design first-class entities encapsulating control and featuring degrees of reactivity and proactivity. This need is particularly relevant as soon as we consider the programming of complex actors or agents, whose behaviour could be articulated and then it becomes important to have clear principles and mechanisms fostering properties in terms of modularity, encapsulation, extensibility, abstraction. This importance can be recognised in particular in the practice, where these models and technologies – actor-based in particular – are more and more adopted in the mainstream as an alternative to multi-threaded programming to develop concurrent/distributed/reactive programs.

Accordingly, the contribution of this paper is, first, to provide a common abstract and informal description of loop-based control architectures spanning from actors (Section 2) to agents (Section 3), considering state-of-the-art languages and technologies, in order to ease the discussion of their properties and their comparison; then, to provide a first discussion of the impact on programming, analysing some features and the drawbacks that – we believe – depend on the control architecture adopted.

2. Actors based on Event-Loops

In literature and in the practice, there are two basic ways to implement actors: without an explicit receive – like in the original model – and with an explicit receive. Examples for the former case include ActorFoundry [18], SALSA [30], Akka [1]. Examples for the latter case include Erlang [2] and Scala Actors as defined in [12]. Even if these could be considered equivalent from the computational model point of view, the two different programming models lead to actor programs with a quite different organization and shape.

Event-loops are the main approach adopted to define the control architecture of actors in the former case. The behaviour of an actor can be abstractly represented by an infinite loop (Algorithm 1) composed essentially by three main stages:

Algorithm 1 Abstract Version of a Basic Actor Event Loop

```
1: loop
2:     msg ← WAITFORMSG()
3:     h ← SELECTMSGHANDLER(msg)
4:     args ← GETMSGARGS(msg)
5:     EXECUTEMSGHANDLER(h, args)
6: end loop
```

First, a message is retrieved from the mailbox, when available (line 2); then, a proper handler or method associated to the message is selected (line 3); finally, the selected handler – if any – is executed, before cycling again (line 4-5). Three key points of the model are:

- pure *reactive* behaviour — an actor starts working only if there is a message in the mailbox;
- a macro-step (or *run-to-completion* [29]) semantics — the execution of a message handler is atomic, i.e. can be represented as a computational step atomically changing the internal state of the actor, and its external environment by delivering new messages or creating new actors.
- strict *no-blocking discipline* — handlers cannot block or engage infinite loops: they must be necessarily finite computations manipulating the internal state of the actors and using asynchronous primitives to send messages and create new actors.

From a design and programming point of view, this model promotes a decomposition based on behaviours similar to the state pattern [11]. On the one side, this makes it particularly effective for implementing actors that can be properly modelled as reactive state machines. Transitions in a state are triggered by the receipt of a message and the atomic execution of the message handler represents the effect of the transitions, changing atomically the state. On the other side, the implementation of more activity/procedure-oriented, hierarchical behaviours, possibly involving patterns of synchronous and asynchronous interactions – is quite problematic and calls the adoption of further mechanisms to preserve a level of modularity.

The problem in this case is the fragmentation of the code in handlers, which does not necessarily corresponds to a good modularization from the point of view of organization of the wanted behavior. That is, a designer is forced to decompose the behaviour following the message flows, eventually using *self-sending* of messages to structure articulated, long-term activities without tampering reactivity. This is clearly a programming trick, decreasingly the level of abstraction used to describe the strategy identified at the design level. It produces similar effects to the *goto* for sequential programs [10], tampering program understanding.

As a concrete example, we consider here a well-known problem in concurrent programming, the dining philosophers. Philosophers must feature a behaviour in which repeatedly alternate thinking with eating, and for the latter they need to properly interact with their environment—efficiently acquiring and using their couple of forks in a mutual exclusive way. The simplest solution to avoid deadlocks is to acquire the forks (labelled with a numerical id) always in the same order, so that the $N-1$ philosopher using the forks tagged as $N-1$ (left) and 0 (right) collects first the fork 0 and then, after this succeeded, the $N-1$ one. In a well-modularised solution the behaviour of the philosopher is decomposed in four main parts: thinking, acquiring forks, eating, releasing forks:

```
process Philosopher(Fork f1, Fork f2) {
 loop {
    think()
    acquireForksInOrder(f1,f2)
    eatUsingForks(f1,f2)
    releaseForks(f1,f2)
  }
}
```

The modules should be as loosely coupled as possible and well-separated, so that e.g. thinking and eating should not need to know anything about the strategy to adopt to acquire/realease forks.

Figure 1 shows a solution in ActorFoundry [18] and in Figure 3 in Akka [1], adapted from a version called "dining hakkers" available in the Akka distribution. In both cases, forks are modelled as actors—the Akka version adopts a solution based on busy waiting. In the former, the philosopher behaviour is decomposed into message handlers that correspond to the different states in which the actor can be. In the latter, the philosopher is more explicitly decomposed into state/behaviours, using a become mechanism to make transitions. In both cases, it is evident the fragmentation of the philosopher behaviour into a set of handlers that depends on the message flow.

This problem can be mitigated by the use of *continuations* [13], specifying the message to be replied when sending a request message, that is the handler to be executed when the request has been completed. Figure 2 shows the philosopher in SALSA [30] as reported in [31]. SALSA supports different forms of continuations directly in the language. In this case, it is possible to specify sequences – or patterns – of actions in the same handler In the philosopher, for instance, in the method `gotRight`, if also the right chopstick has been acquired, then first an "eating" message is printed on standard output and release messages are sent to chopstick actors; then a "thinking" message is printed to out-

```
public class PhiloActor extends Actor {
  private int nForksAcquired;
  private ActorName firstFork, secondFork;

  @message public void start(ActorName[] forks,
                    Integer leftFork, Integer rightFork) {
    if (leftFork < rightFork){
      firstFork = forks[leftFork]; secondFork = forks[rightFork];
    } else {
      firstFork = forks[rightFork]; secondFork = forks[leftFork];
    }
    send(this.self(), "think");
  }
  @message public void think(){ send(this.self(),"hungry"); }
  @message public void hungry(){
    nForksAcquired = 0; send(firstFork, "acquire",this.self());
  }
  @message public void gotFork(){
    nForksAcquired++;
    if (nForksAcquired < 2){
      this.send(secondFork, "acquire",this.self());
    } else {
      this.send(self(), "eat");
    }
  }
  @message public void eat(){ send(self(), "sated"); }
  @message public void sated(){
    send(firstFork, "release",this.self());
    send(secondFork, "release",this.self());
    send(self(), "think");
  }
}
```

Figure 1. A Dining Philosopher in ActorFoundry.

```
behavior Philosopher {
  Chopstick left, right;

  Philosopher(Chopstick left, Chopstick right){
    this.left = left; this.right = right;
  }
  boolean pickLeft(){ left <- get(self) @ currentContinuation; }
  boolean pickRight(){ right <- get(self) @ currentContinuation; }
  void eat(){
    pickLeft() @
    gotLeft(token);
  }
  void gotLeft(boolean leftOk){
    if (leftOk) {
       pickRight() @
       gotRight(token);
    } else eat();
  }
  void gotRight(boolean rightOk){
    if (rightOk) {
      join {
        standardOutput <- println ("eating...");
        left <- release();
        right <- release();
     } @ standardOutput <- println ("thinking...") @
       eat();
     } else gotLeft(true);
  }
}
```

Figure 2. A Dining Philosopher in SALSA, as reported in [31].

```
class Hakker(name: String, left: ActorRef,
                                right: ActorRef) extends Actor {
  import context._

  // thinking behaviour
  def thinking: Receive = {
    case Eat =>
      become(hungry)
      left ! Take(self)
      right ! Take(self)
  }

  def hungry: Receive = {
    case Taken(`left`) =>
      become(waiting_for(right, left))
    case Taken(`right`) =>
      become(waiting_for(left, right))
    case Busy(fork) =>
      become(denied_a_fork)
  }

  def waiting_for(forkToWaitFor: ActorRef,
                           otherFork: ActorRef): Receive = {
    case Taken(`forkToWaitFor`) =>
      become(eating)
      system.scheduler.scheduleOnce(5.seconds, self, Think)
    case Busy(fork) =>
      otherFork ! Put(self)
      startThinking(10.milliseconds)
  }

  def denied_a_fork: Receive = {
    case Taken(fork) =>
      fork ! Put(self)
      startThinking(10.milliseconds)
    case Busy(fork) =>
      startThinking(10.milliseconds)
  }

  // eating behaviour
  def eating: Receive = {
    case Think =>
      left ! Put(self)
      right ! Put(self)
      startThinking(5.seconds)
  }

  // All hakkers start in a non-eating state
  def receive = {
    case Think =>
      startThinking(5.seconds)
  }

  private def startThinking(duration: FiniteDuration): Unit = {
    become(thinking)
    system.scheduler.scheduleOnce(duration, self, Eat)
  }
}
```

Figure 3. A Dining Hakker (Philosopher) in Scala using the Akka framework, as reported in Akka distribution.

put, and finally the cycle is started again by self-sending an eat message.

2.1 Integration with OOP and Impact on Modularity

A further aspect that impacts on the programming of actors based on event-loops is the model adopted to integrate actors and objects. In the model underlying the approaches discussed in previous subsection, actors and objects are essentially two independent levels. It is true that in OOP frameworks like ActorFoundry and Akka, actors are implemented in terms of OOP API – so, for instance, defining actors by extending existing library classes or existing actor classes. However this is just an implementation-level choice: the two levels are conceptually independent and objects are solely used to represent the data structures that are manipulated and exchanged by actors.

A different integrating approach has been introduced with the VAT model [21], where actors (called VAT) are containers of objects and message passing, at the programming level, occurs among the objects themselves. Besides the E language [21], the model has been adopted also by other actor-based language/systems such as AmbientTalk [8], and approaches based on active objects such as JCoBox [27] and ABS [16], extending the basic Creol model [15].

This choice strongly impacts on the way in which the behaviour of an actor is designed and programmed. The behaviour of an actor is decomposed in terms of objects, directly exchanging messages with other objects, possibly hosted in other actors. When created, the actor hosts a single object (called actor's behaviour), functioning as a public interface to the actor, whose *far* reference is returned to the actor creator. The event-loop in this case is a refinement of the basic one (see Algorithm 2), where a heap is used to keep track of the objects inside the actor and message dispatch occurs by first locating the object target of the message, and then invoking the corresponding method, which is run to completion.

Algorithm 2 Abstract Version of an Actor Event Loop

loop
    $msg \leftarrow$ WAITFORMSG()
    $o \leftarrow$ LOCATEOBJECT(msg)
    $m \leftarrow$ GETMETHOD(msg)
    $args \leftarrow$ GETMETHODARGS(msg)
    CALLMETHOD($obj, m, args$)
end loop

The processing of an asynchronous message to completion, that may involve a chain of synchronous method calls among objects inside the hosting actor, backed by a stack – is called *turn*.

In this model the promoted programming style is more similar to the classic OOP one, integrated with asynchronous message passing and strictly no-blocking behaviour. Continuation Passing Style (CPS) is heavily adopted as mechanism to manage asynchronous computations, based on non-blocking futures. This leads to a quite different shape for actors, compared to the one based on states/behaviours. As an example, Figure 4 shows a snipper of the the philosopher implemented in AmbientTalk, available in AmbientTalk distribution. Here a single room actor is used to manage forks, that are collected with a single request. The plan of the overall behaviour is encapsulated in the method `live`, where futures and continuations are used to manage the interactions first to get the forks, and then to eat, and finally to release the forks and start again the cycle.

On the one side, compared to behaviour based actors, this programming style reduces fragmentation, making it possible to encapsulate in a method of an object inside an actor the application logic possibly involving articulated interaction – synchronous and asynchronous – with other objects. On the other side, the massive use of CPS and nested futures/callbacks makes programming challenging, as witnessed by other fields where event-loops and CPS are heavily exploited, such as web programming. In the philosopher example, the nested continuations included in the body of the `live` method recalls the “pyramids of doom” raised by nested callbacks in asynchronous programming.

```
actor: { |i,name,room|
  ...
  def live() {
    when: think() becomes: { |doneThinking|
      when: room<-pickUp(i)@FutureMessage becomes: { |forks|
        when: eat(forks) becomes: { |doneEating|
          room<-putDown(i)@OneWayMessage;
          continuation();
          nil;
        }
      }
    }
  };
  def think() { ... };
  def eat(forks) { ... }
  // asynchronous continuation of the 'live' method
  def continuation := { self<-live() };

  live();
}
```

Figure 4. A snippet of a Dining Philosopher implemented in AmbientTalk (AT).

A further, more methodological-oriented, reflection is about the principles and guidelines for designing programs, in this case. In previous cases, actors are the main blocks to be used to structure a program, each further decomposable in terms of behaviours or states. Here instead, passive objects – and not actors – are the fine-grained blocks to be used, like in OOP – in fact, every interaction is among objects and the communication between actors must be explicitly conceived as the communication between objects inside them. So a question is: what's the methodology and principle that could drive the design of programs in this case. It is not purely actor-based, but it is not even pure OO, since here we deal with concurrency and distribution as first-class aspects. This could be subject of further investigation and development in literature.

3. From Event-Loops to Control-Loops

In literature, control-loops have been introduced and adopted in different contexts – from control theory, to AI and software engineering – in particular to define the control architecture of autonomous/autonomic components (devices, agents, robots,..) interacting with some kind of environment. In the case of autonomic computing [19], the control-loop ruling the behaviour of an autonomic component is called MAPE and is composed by four conceptual stages which are repeatedly executed – Monitor, Analyse, Plan and Execute. In the case of Agent-Oriented Programming as defined originally in [28], it is used to define the abstract architecture of intelligent agents, so as to bring together proactivity

– that is, acting towards the achievement of some goal – and reactivity – to promptly react to relevant events occurring in the environment. The *reasoning cycle* of BDI intelligent agents [23] is a control-loop, based on three macro-stages – *sense, plan, act* – that are executed at each cycle of the loop. Different variants of this cycle have been implemented in practical agent programming languages/frameworks, such as Jason [4], AgentFactory [26], 2APL [9], GOAL [14]—all these mainly in the context of Distributed Artificial Intelligence.

Besides that context, in our previous works we started exploring the value of this kind of control-loop also for defining the behaviour of agents adopted as fine-grained first-class abstractions in concurrent and distributed programming, comparable to actors. Agents in the simpAL [25] language and in its most recent evolution called ALOO [24] embed a simplified variants of the BDI reasoning cycle, sharing many characteristics and features with actors' event-loops.

Said this, in the following we discuss the properties of control-loops from a programming point of view, taking Jason and ALOO as reference cases..

Algorithm 3 Simplified version of the AgentSpeak(L)/Jason control-loop

1: $B \leftarrow B_0$; $PlanLib \leftarrow PlanLib_0$; $Ev \leftarrow \{\}$; $I \leftarrow \{\}$
2: **loop**
3:     $\rho \leftarrow$ SENSEENV()
4:     BELUPDATE(ρ, B, Ev)
5:     **if** Ev is not empty **then**
6:         $ev \leftarrow$ FETCHEVENT(Ev)
7:         $p \leftarrow$ SELECTPLAN($ev, B, PlanLib$)
8:         **if** ev is an env change or a new goal to achieve **then**
9:             $I \leftarrow I \cup \{$NEWINT(p, ev)$\}$
10:         **else if** ev is a sub-goal to achieve **then**
11:             PUSHPLAN($currInt, p, ev$)
12:         **end if**
13:     **end if**
14:     **if** I is not empty **then**
15:         $currInt \leftarrow$ SELECTINTENTION(I)
16:         $a \leftarrow$ FETCHNEXTACTION($currInt$)
17:         EXECACTION($a, currInt, B, I, PlanLib$)
18:     **end if**
19: **end loop**

3.1 Control-Loops in Intelligent Agents

Jason is a concrete implementation and extension of the AgentSpeak(L) language [22]. It has been conceived to be a practical language to implement intelligent BDI-based agents, adopting Prolog and logic programming as background language to represent data structures. An agent in Jason is an autonomous entity – owning a logical control flow – functioning as a *reactive planning system*, reacting to events perceived from the environment where it is immersed and doing actions on that environment in order to achieve some assigned goal(s), possibly exchanging messages with other agents. It is programmed in terms of *goals* – representing the tasks that can be allocated to the agent – *beliefs* – Prolog-like facts and rules used to represent the internal/hidden agent state, including the information about the current perceived state of the environment – and *plans* – which encapsulate the procedural knowledge to be used to react to events like changes to beliefs, new goals to achieve or goal failures. Details about Jason programming model can be found here [3, 4].

Algorithm 3 shows an abstract simplified version of the Jason control-loop. The knowledge state of the agent is represent by its belief base B, which contains both facts about private state of the agent and its beliefs about the current observable state of the environment. The plan library $PlanLib$ contains the description of the set of plans that the agent can use to achieve goals. I is the set of agent ongoing *intentions*, i.e. the ongoing plans in execution (that corresponds to the concrete activities that the agent is carrying on).

Lines 3-4 represent the sense stage of the loop. The current state of the environment is perceived in terms of a set of percepts ρ, and these are used to update the belief base of the agent, i.e. its beliefs about the state of the environment. In this simplified version, these percepts include also the messages sent by other agents. The updates of the belief base generate a new set of events which are added to set of events Ev. In Jason, events in general can be related either to a change in the environment, or a new goal/subgoal to achieve – either self-allocated or allocated by another agent. The plan stage is given by lines 5-13. If the set of events is not empty, then one event ev is fetched and a relevant and applicable plan p is selected for dealing with the event ev, given current agent beliefs B and current set of plans available in the plan library $PlanLib$. Then, a new intention is added to the intention set I if the event ev is about an environment change or a new independent goal to achieve. Otherwise, if the event is about a sub-goal to achieve, meaning that it is a sub-goal of current plan in execution, then the plan p is pushed on the plan stack associated to the current intention. The effect is similar to a procedure call: the current plan in execution is suspended until the sub-goal is achieved—that is, the new plan p is completed with success. Finally, in the act stage (lines 14-18), one intention among the ongoing intentions I is scheduled to be the current intention $currInt$, from which the next action a to be executed is fetched. The action is executed—affecting either the agent state (if it is an internal action) or the environment (if it is an external action).

By comparing the control-loop implemented in Jason and event-loops discussed in previous section, it is possible to recognise some important similarities. In particular, the sense stage in this control-loop corresponds to message fetching in event-loops, the plan stage to message handler selection and the act stage to handler execution. Like in event-loops, also in control-loops the following features hold:

- no low-level race conditions can occur inside the agent, since there is only one logical control flow accessing/-modifying the agent state. This access is staged, so that in the sense stage it is updated, in the plan stage it is read, and finally in the act stage it is read or updated, depending on the action executed;
- no low level deadlocks can occur since the control flow executing the cycle is never blocked. Even when the execution of a plan is suspended because an agent is waiting for e.g. the completion of some action to execute the next one, it can always react to other events, instantiating new intentions.

Besides, there are some differences that – we will see – have a relevant impact on programming. In particular, in control-loops:

- the run-to-completion semantics is relaxed, in fact plans selected in the plan stage (and corresponding new created intentions) are not necessarily run to completion before the next cycle, but through multiple cycles, executing only one atomic action at each cycle and possibly managing multiple intentions;
- if there are no events to process, the cycle is not blocked: it may go on selecting and performing actions following the current intention(s);
- a preemptive scheduling schema is adopted to carry on multiple plans—instead of a cooperative one, as adopted in container-based actors and active objects. However, like in actors, as already said, no low level race condition can occur, since individual actions are executed atomically;
- a plan in execution can be blocked or suspended—this happens each time an environment/external action is executed (waiting for its completion before executing the next one) or by means of predefined `.suspend` internal actions that allows for suspending ongoing intentions. However, as mentioned before, the agent per se is not blocked: the agent execution cycle is always running, eventually reacting to events relevant for the agent.

From a programming model point of view, this kind of control-loop promotes a decomposition of the behaviour based on *plans* as hierarchical procedural abstractions, so a quite different approach with respect to the state/behaviour-based one promoted by the basic actor model. As an example, Figure 5 shows the dining philosopher implemented in Jason. The behaviour is given by a set of plans, logically organized in a hierarchical fashion. The agent reacts to the initial goal `!boot(F1,F2)`, by instantiating a plan (lines 1-3) in which first it sorts out the forks to be used (`!sort_forks(...)` sub-goal) and then starts an end-less living activity (`!!living` independent goal). In Jason, `!`*G* and `!!`*G* specify respectively a subgoal and an independent goal *G* to be achieved, the latter case creating a new

```
1  +!boot(F1,F2)
2    <- !sort_forks(F1,F2);
3       !!living.
4
5  +!sort_forks(F1,F2) : F1 <= F2
6    <- +first(F1); +second(F2).
7
8  +!sort_forks(F1,F2) : F1 > F2
9    <- +first(F2); +second(F1).
10
11 +!living
12   <- !think;
13      !acquireRes;
14      !eat;
15      !releaseRes;
16      !!living.
17
18 +!acquireRes : first(F1) & second(F2)
19   <- acquireFork(F1); acquireFork(F2).
20
21 +!releaseRes: first(F1) & second(F2)
22   <- releaseFork(F1); releaseFork(F2).
23
24 +!think <- println("Thinking").
25 +!eat <- println("Eating").
```

Figure 5. A Dining Philosopher implemented in Jason.

intention. The plan for living (lines 11-16) accounts for a sequence of subgoals, first think, then acquire forks, eat, and then release forks before starting again with the same activity. In the plan for acquiring the forks (lines 18-19), the agent interacts with the environment by performing the `acquireRes` actions in order. The shape of the resulting program is as simple as the one typically found in multi-threaded programming, even if here the control architecture is completely different.

On the one side, this model makes it more natural compared to event-loop based approaches the design and implementation of activity/process oriented behaviours that need to integrate some kind of reactivity. On the other side, this requires a more complex control architecture, in which e.g. a stack must be be used for each plan in execution. Event-loop based actors may use a single stack to manage synchronous method calls among objects in a single turn—however the model ensures that the stack is always empty when the loop is going to wait for the next message to be served.

A simple example of integration between proactivity and reactivity is given by an extension of the dining philosopher problem with a further specification that, besides repeatedly thinking and eating, a philosopher must be able to promptly react to an alarm notified by the environment and then starting an evacuation plan. In Jason, this could be done quite flexibly by extending the agent program with a couple of further plans:

```
+alarm <- .drop_all_intentions; !evacuate.
+!evacuate <- ...
```

in which the agent reacts to new belief `alam` perceived from the environment (that could be replaced by a message sent by another agent), and then drops all ongoing intentions (`.drop_all_intentions` is a primitive internal action) and instantiates an new `!evacuate` goal. Apparently, this kind of flexibility is not that easy to be achieved in basic event-loop based models—where it could require to explicitly modify each actor state/behaviour to add an handler to react to the alarm message. Instead, this could be done more easily in container-based event-loops, since the alarm message could be sent to some specific object inside the actor, without the need to change the other objects hosted by the same actor container.

3.2 Control-Loops for Agents in Object-Oriented Concurrent Programming

The capability of flexibly integrating reactivity and proactivity makes this kind of control-loop interesting also in the context of concurrent programming, where the integration of event-driven/asynchronous and thread-oriented/synchronous programming is still an issue today. Accordingly, we adopted a simplified version of the sense-plan-act control-loop as control architecture of agents in simpAL [25] and in the more recent ALOO [24] language.

In ALOO in particular the objective is explore the extension of classic sequential object-oriented programming with an agent-oriented abstraction layer to address concurrency and featuring agents with a minimal sense-plan-act loop. In ALOO, objects are used to model any kind of passive entity / data structure which is dynamically created, possibly shared and used by agents. Agents, on the other hand, are used to model fine-grained active entities in charge of fulfilling some task, by dynamically using and observing objects. Fine-grained means that a program in execution can host as many agents as objects—they are like lightweight actors/processes in languages such as Erlang. The set of objects represent the environment where agents are logically situated, providing them the actions that they can do – corresponding to object operations – and perceptions – corresponding to changes to object observable properties. Differently from objects, agents don't need to be garbage collected: they terminate as soon as (if) they complete their task.

The detailed description of ALOO programming model is out of the scope of this paper: in the following we provide just the essential elements that are useful to support the discussion. To help this description, we consider the source code of a philosopher agent in ALOO, shown in Figure 6, that will be discussed also after presenting the control loop.

The structure and behaviour of an agent in ALOO is defined by *agent scripts* (e.g. `Philosopher` agent script in Figure 6), similar to classes, containing a set of *plans* and of variables defining the global agent state, shared by plans in execution (e.g. `first` and `second` variables, in the `Philosopher` agent script). Like in Jason, plans contain the recipes that the agent can use to achieve its task(s), and the term *intention* is used to refer to a plan in execution. At runtime, an agent is created with a task to do and it terminates when/if it completes the main intention created to fulfill that task.

```
1  task DiningTask {
2    leftFork, rightFork: Fork
3    alarm: boolean
4  }
5
6  agent-script Philosopher {
7    public-tasks: DiningTask;
8    Fork first, second;
9
10   plan-for DiningTask {
11     if (this-task.leftFork.id < this-task.rightFork.id){
12       first = this-task.leftFork; second = this-task.rightFork
13     } else {
14       first = this-task.rightFork; second = this-task.leftFork
15     };
16     {
17       [] => {
18         do Think();
19         do AcquireForks();
20         do Eat();
21         do ReleaseForks()
22       }
23     }
24   }
25   plan-for Think() { this-env.out.println("Thinking") }
26   plan-for Eat() { this-env.out.println("Eating") }
27   plan-for AcquireForks() { first.acquire() ; second.acquire() }
28   plan-for ReleaseForks() { first.release() ; second.release() }
29 }
```

Figure 6. A Dining Philosopher implemented in ALOO.

A plan (e.g. lines 10-24) is defined by the type of tasks for which it can be used (e.g. `DiningTask`) and a body, specifying how to achieve that kind of tasks[1]. The body of a plan is given by a set of *action rules* and local variables, structured in *blocks* `{ ... }`, defining their scope. Action rules[2] drive the selection of actions[3], specifying *when* an action can be collected in a cycle to be executed. Action rule blocks can be

[1] Tasks in ALOO are uniformly represented by objects, as instances of classes whose interface (type) must be an extension of a predefined `Task` interface. To shorten the declaration and denotation of task objects, some syntactic sugar is provided. The construct `task T { ... }` (e.g. lines 1-4) implicitly defines an interface `T` extending `Task` with a corresponding default class implementing `T`. Objects in ALOO extends normal objects with observable properties, which are declared in the interface along with operation signatures. So `DiningTask` objects (as defined in lines 1-4) have three observable properties: `leftFork`, `rightFork`, and `alarm`.

[2] Each action rule has the general form `[ +e | c ] => a #l`, specifying that the action *a* – labelled as *l* – can be selected to be executed each time an event *e* occurs and the condition over agent state *c* holds. Events are related either to changes to observable properties of objects that the agent is observing or changes to the execution state of actions of the block. Either the event or the condition can be omitted, meaning that the action can be selected independently from – respectively – the happening of some specific event or from the current action state. The `[] => a` rule (e.g. lines 19-25) means that the action *a* can be always selected. Some syntactic sugar is provided to directly encode set of rules representing sequence of actions: they can be written as a chain of actions (omitting the condition part) using the semicolon separator, like a sequence of statements in the case imperative programs (e.g. lines 20-25).

[3] Actions can be external – i.e. invoking method on objects, given their references – or internal, e.g. assigning a value to a variable. Actions related to method execution are carried asynchronously, by a different control flow from the control-loop; the completion or failure of actions is perceived by the agent as asynchronous events. Among the internal actions, the `do` action instantiate a new sub-task to be achieved (e.g. line 20, 21, 22, 23), specifying the object that represents the new task to accomplish. The `do`

nested, by specifying actions that are blocks themselves[4]. Finally, the definition of an action rule block can include also the declaration of those (object) observable properties that the agent needs to observe inside the block[5]—the updated value of observed observable property is stored in local read-only variables called *beliefs* that can be accessed inside the block. Beliefs are used also to keep track of the execution state of actions.

The control-loop in ALOO is shown in Algorithm 4. S

Algorithm 4 ALOO control-loop

1: $S \leftarrow S_0$; $PlanLib \leftarrow PlanLib_0$; $Ev \leftarrow \{\}$
2: $p \leftarrow$ SELECTPLAN$(AssignedTask, PlanLib)$
3: $I \leftarrow \{$NEWINT$(p, AssignedTask)\}$
4: **while** I is not empty **do**
5: $\quad currInt \leftarrow$ SELECTINTENTION(I)
6: $\quad ev \leftarrow$ FETCHEVENT$(Ev, currInt)$
7: $\quad$ UPDATEBEL$(currInt, ev)$
8: $\quad$ **if** ev is about a new sub-task t todo **then**
9: $\qquad p \leftarrow$ SELECTPLAN$(t, PlanLib)$
10: $\qquad$ PUSHPLAN$(currInt, p, t)$
11: $\quad$ **else if** ev is about a new task t todo **then**
12: $\qquad p \leftarrow$ SELECTPLAN$(t, PlanLib)$
13: $\qquad I \leftarrow I \cup \{$NEWINT$(p, t)\}$
14: $\quad$ **end if**
15: $\quad a_l \leftarrow$ COLLECTACTIONS$(currInt, S, ev)$
16: $\quad$ **for all** a in a_l **do**
17: $\qquad$ EXECACTION$(a, currInt, S, I, PlanLib)$
18: $\quad$ **end for**
19: **end while**

represents the agent global state (variables), I the set of ongoing plans in execution (i.e., intentions), Ev is the event queue, $PlanLib$ the plan library, storing the current set of plans available to the agent (loaded from agent scripts), $AssignedTask$ the reference to the object representing the task assigned to the agent.

In this control-loop, there is a first *plan* stage before looping (lines 2–3), selecting a plan p for the assigned task and then instantiating the corresponding intention in the set of intentions I. Then, the loop is used to carry on the execution of the plan (act stage, lines 15–18), while perceiving events from the environment (sense stage, lines 5–7). Similarly to event-loops, for each cycle an event is fetched from the event queue (line 6). Like in Jason and differently from event-loops, such fetching is not blocking: if no events are in the queue, ev is nil (not available). Differently from Jason, fetching is driven by the intention, that is: fetching looks for an event related to the current intention $currInt$—i.e., concerning beliefs belonging to the action rule block on the top of the stack of this intention. If the event is about the change of an observable property of an object observed by the agent, or about the notification that an action previously executed (e.g. method call on an object) has been completed (or failed), then the corresponding belief in the action rule block is updated (line 7). If the event is about a new task to do – caused by the execution of an action self-allocating a task, such as the do action – then a new plan stage is executed (line 8-14). Like in Jason, if the request is about a sub-task to do, then the plan body of the selected plan (which is an action rule block) is pushed on current intention stack. Otherwise, if it is an independent task, a new intention is created. In the act stage, an intention is selected to be the current intention $currInt$ (line 7), using a round-robin schema to guarantee fairness, and then all actions that can be executed according to the rules of such an intention are collected (line 15) and executed, sequentially (line 16-18).

Two important differences compared to the control-loop in Jason are:

- here the plan stage does not occur for every possible event, but only for events concerning the tasks to do—reactions to events related to changes in the environment are expressed by action rules inside plans;
- it is not an infinite loop: an agent terminates as soon as there are no more intentions to carry on.

The programming model induced by this control-loop is quite similar to the Jason one, based on the hierarchical decomposition of plans, as shown by the philosopher example in Figure 6. The main differences concern the granularity of plans and the shape of the behaviours integrating proactivity and reactivity. Jason – following AgentSpeak(L) – adopts a quite fine grained plan model, so that we have to write a plan for each possible event relevant to the agent. On the one side this favors simplicity and flexibility, on the other side it has a drawback on modularity and encapsulation: a strategy for doing some task that needs to integrate some actions and some reactions to asynchronous events cannot be encapsulated into a single plan, but it must be necessarily split into multiple plans not explicitly related. So for complex agent programs, this could lead to a large number of plans which are not sub-plans but implicit fragments of the same logical high-level plan.

For instance, suppose to consider a variant of the dining philosopher in which a philosopher agent must begin eating reactively, by perceiving a `hungry` stimulus after thinking. This could be implemented in Jason as follows:

```
+!living <- !think.
+hungry <- !acquireRes; !eat; !releaseRes; !!living.
```

that is: the plan for `!living` must be broken in two parts: a first plan triggering the `!think` goal and a second plan reacting to the event `+hungry`. At the logical level, these

syntax allows to specify directly the name of the task type, along with parameters—in that case, a new task object of that type is implicitly created.

[4] So for each intention, a stack is used to manage the action rule block nesting.

[5] Given an object `o` with an observable property `obs`, then in an action rule block `|o.obs as: b|{ ... [+b] => ... }` the agent can perceive changes occurring to `o.obs`, mapped into an implicitly declared local read-only variable `b`. These variables are called *beliefs*.

plans are part of the same conceptual higher-level plan, but they are not explicitly related at the program level.

The ALOO control loop allows for adopting a more coarse-grained plan model: a plan is meant to encapsulate the strategy to accomplish some specific task—which may include some workflow of actions – including triggering further sub-tasks – and reactions. For instance, the variant of the dining philosopher could be implemented without breaking the plan, by simply adding an action rule:

```
agent-script Philosopher(body: MyBody){
 plan-for DiningTask {
  ...
  {
    [] => |body.isHungry as: hungry| {
      do Think()
      [hungry] => {
        do AcquireForks();do Eat();do ReleaseForks()
      }}}
```

In this code, the agent in the main cycle observes the observable property `isHungry` of a `body` object, mapped into a belief `hungry`. As soon as it is perceived to be true, the agent executes an action rule block driving the eating stage. This allows to avoid the enforced fragmentations into plans to handle reactivity, improving encapsulation—at the price of an increased complexity of the plan model adopted and of the structures used to manage it at runtime. gaz

4. Discussion and Concluding Remarks

Given the analysis in previous sections, we can draw a path from threads to actors to agents concerning the control architecture adopted to define the behaviour of autonomous entities. With threads, the control architecture accounts for a simple control flow executing some body of code, possibly traversing objects shared with other control flows. With receive-based actors (not based on event-loop), such a control flow is encapsulated into boundaries so that it cannot cross with other control flows, it can traverse objects that are inside these boundaries and an explicit blocking receive primitive is provided to react to messages. With event-loop based actors, the control architecture is extended to provide a stronger discipline: it allows the programmer for abstracting from the use of low-level receive primitives by organising the execution flow in turns or cycles and embedding the blocking receive as an implicit part of the control architecture. This promotes a state-based organisation of the actor behavior; then, the adoption of mechanisms such as continuations and futures/promises allows for partially recovering a more activity-oriented style. Going from event-loops to control-loops, the implicit blocking behaviour of the control flow is removed, so that conceptually the cycle is continuously running, fetching an event if available at each cycle, deciding the next action(s) to do according to the current plan(s) in execution and executing it/them. This leads to plan/procedural organization of the behaviour, allowing for integrating blocking actions without tampering reactivity. The increased complexity of the control architecture corresponds to an increase of the level of abstraction provided by the programming model, where e.g. mechanisms such as continuations are no more necessary to manage synchronous interactions or to realize articulated activities. So what is quite clear in this path is that the evolution of the level of abstraction provided to program active entities is strongly related to the evolution of the control architecture adopted.

This analysis can be considered just the starting point, useful to set a frame – spanning from event-loops to control-loops, from actors to agents – in which (further) issues can be (further) explored. One is about performance, i.e. how the control architecture may impact on performances. The evolution of the control architecture discussed before corresponds to an increase of complexity and challenges about performances. In particular, the features provided by control-loop in agents could lead to an important decrease of performance compared to event-loops in actors. This could be devised also by considering some first benchmarking and analysis recently carried on in literature [6]. However, the level of maturity of agent technologies is far from the actor one, where different kinds of optimisation have been devised and applied making the performance of event-loop based actors comparable with the one based on explicit receive [18]. Analogously, we believe that a deeper study of control-loop implementation schema can lead to optimizations making the performance of agents comparable to actors one. An example is given by *cycling-by-need*, that is: even if the control-loop is, in principle, always cycling, without blocking, it is possible to identify those situations in which cycling can be avoided, since it is not going to change the agent actual state.

Another issue that can be further explored, which could be considered as a direct continuation of this work, is about the impact of control architectures on the mechanisms and models that can be adopted to extend/reuse/compose the structure and behaviour of actors/agents. A simple example which is mentioned also in this paper is about how easy can be extending the task/behaviour of a dining philosopher with the capability/functionality of promptly reacting to an alarm stimoulous/message and evacuate, starting from the pre-existing basic implementation. Apparently, this problem can be tackled in different ways depending on the kind of organization of the behaviour promoted, e.g. either in states or in plans. In actor literature, most of the discussion about reuse and extensibility has been developed around the problem of inheritance anomaly [20], focusing in particular on purely reactive entities (the typical example is a bounded buffer actor). This could be the starting point to consider also more proactive entities, like agents, and organization of behaviours that are more plan-oriented, integrating existing research works about inheritance available in the context of agent programming [7, 17].

References

[1] Akka. Actor framework, 2010. `http://akka.io/`, Last Retrieved: Aug 12, 2014.

[2] J. Armstrong. Erlang. *Communications of the ACM*, 53(9):68–75, 2010.

[3] R. Bordini and J. Hübner. BDI agent programming in AgentSpeak using Jason. In F. Toni and P. Torroni, editors, *CLIMA VI*, volume 3900 of *LNAI*, pages 143–164. Springer, Mar. 2006.

[4] R. Bordini, J. Hübner, and M. Wooldridge. *Programming Multi-Agent Systems in AgentSpeak Using Jason*. John Wiley & Sons, Ltd, 2007.

[5] Y. Brun, G. Marzo Serugendo, C. Gacek, H. Giese, H. Kienle, M. Litoiu, H. Müller, M. Pezzè, and M. Shaw. Software engineering for self-adaptive systems. chapter Engineering Self-Adaptive Systems Through Feedback Loops, pages 48–70. Springer-Verlag, Berlin, Heidelberg, 2009.

[6] R. C. Cardoso, M. R. Zatelli, J. F. Hübner, and R. H. Bordini. Towards benchmarking actor- and agent-based programming languages. In *Proc. of the 2013 Workshop on Programming Based on Actors, Agents, and Decentralized Control*, AGERE! '13, pages 115–126, New York, NY, USA, 2013. ACM.

[7] L. Crnogorac, A. S. Rao, and K. Ramamohanarao. Analysis of inheritance mechanisms in agent-oriented programming. In *IJCAI (1)'97*, pages 647–654, 1997.

[8] T. V. Cutsem, S. Mostinckx, E. G. Boix, J. Dedecker, and W. D. Meuter. AmbientTalk: Object-oriented event-driven programming in mobile ad hoc networks. In *Proc. of SCCC '07*, pages 3–12, Washington, DC, USA, 2007. IEEE Computer Society.

[9] M. Dastani. 2apl: a practical agent programming language. *Autonomous Agents and Multi-Agent Systems*, 16(3):214–248, 2008.

[10] E. W. Dijkstra. Letters to the editor: Go to statement considered harmful. *Commun. ACM*, 11(3):147–148, Mar. 1968.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[12] P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2-3):202–220, 2009.

[13] C. Hewitt and H. j. Baker. Actors and continuous functionals. Technical report, MIT/LCS/TR-194, 1979.

[14] K. V. Hindriks. Programming rational agents in GOAL. In R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, editors, *Multi-Agent Programming: Languages, Platforms and Applications (2nd volume)*, pages 3–37. Springer-Verlag, 2009.

[15] E. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. *Software & Systems Modeling*, 6(1):39–58, 2007.

[16] E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In *Formal Methods for Components and Objects*, volume 6957 of *LNCS*, pages 142–164. Springer, 2012.

[17] H. Jordan, S. Russell, G. O'Hare, and R. Collier. Reuse by inheritance in agent programming languages. In *Intelligent Distributed Computing V*, volume 382 of *Studies in Computational Intelligence*, pages 279–289. Springer Berlin Heidelberg, 2012.

[18] R. K. Karmani, A. Shali, and G. Agha. Actor frameworks for the jvm platform: a comparative analysis. In *Proc. of PPPJ'09*, pages 11–20, New York, NY, USA, 2009. ACM.

[19] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, Jan. 2003.

[20] S. Matsuoka and A. Yonezawa. Research directions in concurrent object-oriented programming. chapter Analysis of inheritance anomaly in object-oriented concurrent programming languages, pages 107–150. MIT Press, Cambridge, MA, USA, 1993.

[21] M. Miller, E. Tribble, and J. Shapiro. Concurrency among strangers: programming in E as plan coordination. In *Trustworthy Global Computing*, volume 3705 of *LNCS*, pages 195–229. Springer Berlin / Heidelberg, 2005.

[22] A. S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In *Proc. of MAAMAW'96*, pages 42–55. Springer-Verlag New York, Inc., 1996.

[23] A. S. Rao and M. P. Georgeff. BDI Agents: From Theory to Practice. In *Proc. of ICMAS'95*, 1995.

[24] A. Ricci and A. Santi. Concurrent object-oriented programming with agent-oriented abstractions: The ALOO approach. In *Proc. of the 2013 Workshop on Programming Based on Actors, Agents, and Decentralized Control*, AGERE! '13, pages 127–138, New York, NY, USA, 2013. ACM.

[25] A. Ricci and A. Santi. From actors and concurrent objects to agent-oriented programming in simpAL. In *Concurrent Objects and Beyond – Festschrift in Honor of Akinori Yonezawa*, volume 8665 of *LNCS*. Springer, 2014.

[26] R. Ross, R. Collier, and G. O'Hare. AF-APL: Bridging principles and practice in agent oriented languages. In *Programming Multi-Agent Systems*, volume 3346 of *Lecture Notes in Computer Science*, pages 66–88. Springer, 2005.

[27] J. Schäfer and A. Poetzsch-Heffter. CoBoxes: Unifying active objects and structured heaps. In *Proc. of FMOODS'08*, pages 201–219, Berlin, Heidelberg, 2008. Springer-Verlag.

[28] Y. Shoham. Agent-oriented programming. *Artificial Intelligence*, 60(1):51–92, 1993.

[29] T. Van Cutsem, C. Scholliers, D. Harnie, and W. De Meuter. An operational semantics of event loop concurrency in ambienttalk. Technical report, Vrije Universiteit Brussel, 2012.

[30] C. Varela and G. Agha. Programming dynamically reconfigurable open systems with salsa. *SIGPLAN Not.*, 36(12):20–34, Dec. 2001.

[31] C. A. Varela. *Programming Distributed Computing Systems: A Foundational Approach*. MIT Press, May 2013.