AGERE! @ SPLASH 2014
Portland, Oregon

FROM ACTOR EVENT LOOP TO AGENT CONTROL LOOP: IMPACT ON PROGRAMMING

ALESSANDRO RICCI,
University of Bologna, Italy

INTRODUCTION

- Event loop and control loop as well-known control architectures
  - OS, web/mobile apps, ...
  - autonomic computing, AI, control-theory…
- With actors and agents => embedded inside main programming abstractions

=> impact on the design & programming of actors and agents

- modularity, encapsulation, abstraction

OBJECTIVE

- Analyse the impact on design & programming given by the loops
  - examples using Dining Philosopher with state-of-the-art actor/agent technologies
    - ActorFoundry, Akka, SALSA, AmbientTalk
    - Jason, ALOO
- Discuss and compare actor vs. agent approaches

ACTORS

- with explicit receive
  - e.g. Erlang, Scala actors,…
- without explicit receive
  - e.g. ActorFoundry, SALSA, AmbientTalk, …
  - event-loop

ACTOR EVENT-LOOP

Algorithm 1 Abstract Version of a Basic Actor Event Loop

```
loop
    msg ← WaitForMsg()
    h ← SelectMsgHandler(msg)
    args ← GetMsgArgs(msg)
    ExecuteMsgHandler(h, args)
end loop
```

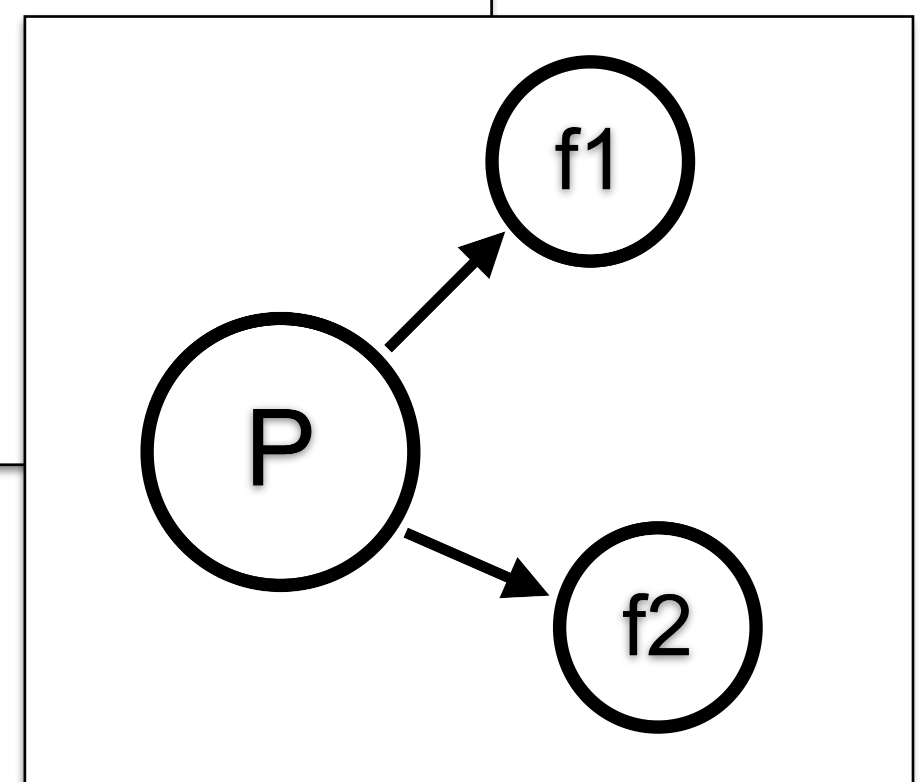
- pure *reactive* behaviour
- macro-step (run-to-completion) semantics
- strict *no-blocking* discipline

IMPACT ON PROGRAMMING

- **Organization**
  - decomposition principle based on messages & message handlers
  - message handlers collected into *behaviors*
- **Pros**
  - effective for modeling/programming reactive state machines
    - states <=> behaviours
    - transitions triggered by messages
- **Cons**
  - quite tricky when dealing with activities
    - procedural/process/task oriented
  - *spaghetti* effect
    - impact on the level of abstraction, program understanding

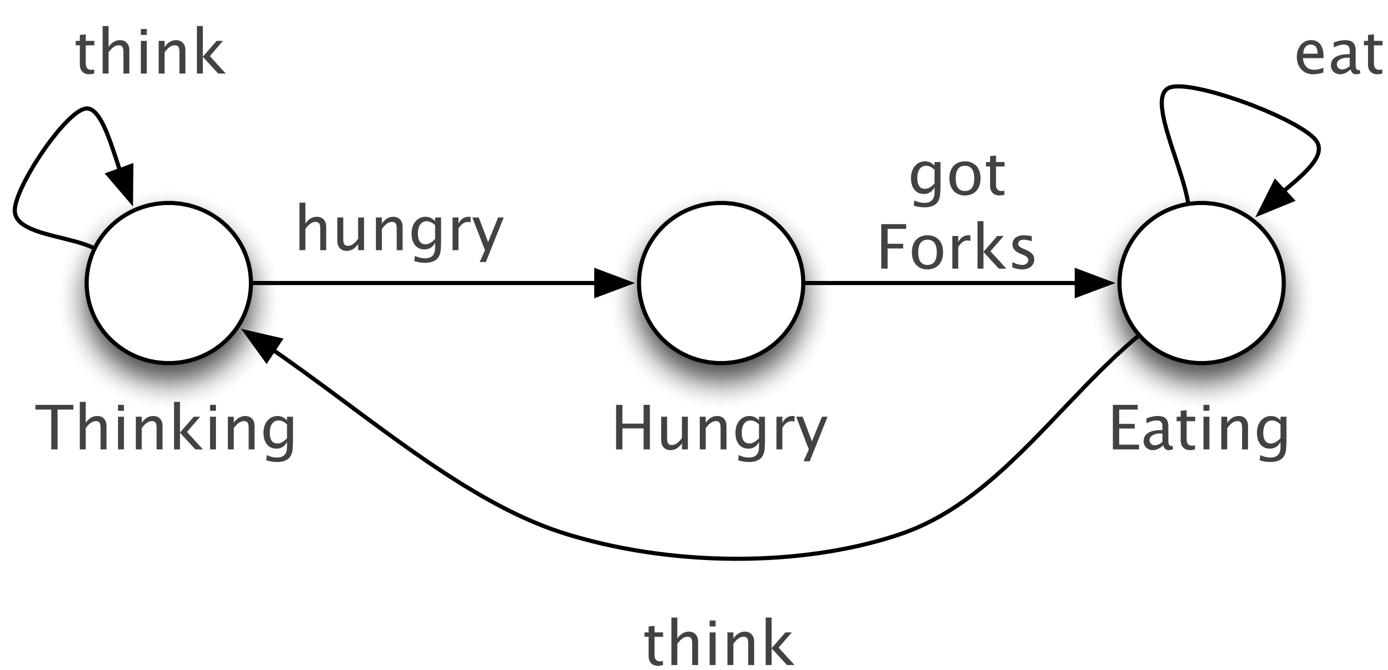
DINING PHILOSOPHER TOY EXAMPLE

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



EXAMPLE: DP IN AF

```
public class PhiloActor extends Actor {
  …
  @message public void start(ActorName[] forks,  Integer f1, Integer f2) {
    … send(this.self(), “think"); }
  @message public void think(){ … send(this.self(),"hungry"); … }
  @message public void hungry(){ … send(firstFork, "acquire",this.self()); … }
  @message public void gotFork(){ … this.send(self(), "eat"); …}
  @message public void eat(){
   … send(firstFork, “release”,this.self());… send(self(), “think”);… }
}
```

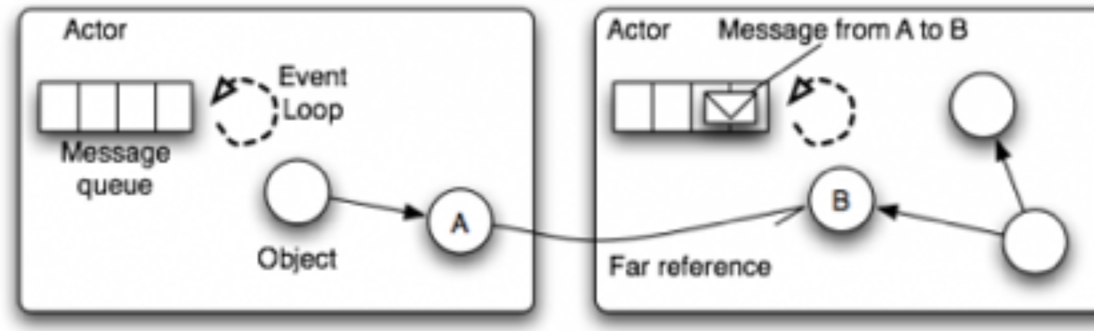


IN THE PAPER

- Exploting behaviors
  - Akka example
- Exploiting continuations
  - SALSA example

INTEGRATION WITH OOP

- event-loop actors + OOP
  => strong impact on program design
  - a main example: VAT model
    - actors as containers of objects
    - languages: E, AmbientTalk,...



Algorithm 2 Abstract Version of an Actor Event Loop

loop
    $msg \leftarrow \textsc{WaitForMsg}()$
    $o \leftarrow \textsc{LocateObject}(msg)$
    $m \leftarrow \textsc{GetMethod}(msg)$
    $args \leftarrow \textsc{GetMethodArgs}(msg)$
    $\textsc{CallMethod}(obj, m, args)$
end loop

AMBIENT-TALK EXAMPLE

```
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) { ... }
 def continuation := { self<-live() };
 live();
}
```

https://github.com/AmbientTalk/AmbientTa

- heavily based on futures & continuation passing style

Pros

- reduced fragmentation
- seamless integration with OOP style and philosophy

Issues

- CPS problems
  - *pyramid of doom*
- design principles? (actors, objects,..)

AGENT CONTROL LOOP

- Control architecture of agents
  - autonomy + goal/task orientation + reactivity

```
loop
  msg := waitForMsg()
  h := selectHandler(msg)
  execute(h)
```

→

```
loop
  sense()
  plan()
  act()
```

- Contexts
  - Agent-Oriented Programming for Intelligent Agents
    - reasoning cycle of BDI models/architectures
    - lang: AgentSpeak(L)/**Jason**, AgentFactory, 2APL, GOAL…
  - Agent-oriented general-purpose concurrent programming
    - examples: simpAL, **ALOO**

IMPACT ON PROGRAMMING

- **Organization**
  - decomposition principle based on *tasks* (what) and *plans* (how)
    - plans encapsulate the strategy for proactively achieving the task, eventually reacting to environment events/input
- **Pros**
  - effective for modeling/programming structured/hierarchical activities
  - integrating pro-active and reactive behaviour
- **Cons**
  - complexity of the loop
  - performance

REASONING CYCLE IN JASON

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

```
B ← B0; PlanLib ← PlanLib0 ; Ev ← {} ; I ← {}
loop
    ρ ← SENSEENV()
    BELUPDATE(ρ, B, Ev)
    if Ev is not empty then
        ev ← FETCHEVENT(Ev)
        p ← SELECTPLAN(ev, B, PlanLib)
        if ev is an env change or a new goal to achieve then
            I ← I ∪ {NEWINT(p, ev)}
        else if ev is a sub-goal to achieve then
            PUSHPLAN(currInt, p, ev)
        end if
    end if
    if I is not empty then
        currInt ← SELECTINTENTION(I)
        a ← FETCHNEXTACTION(currInt)
        EXECACTION(a, currInt, B, I, PlanLib)
    end if
end loop
```

SENSE

PLAN

ACT

Differently from event-loops:

- cycle is conceptually *never blocked*
- macro-step at *action level*

Like event-loops:

- no low-level races
- no low-level deadlocks

DP IN JASON

```
+!boot(F1,F2) <- !sort_forks(F1,F2); !!living.

+!sort_forks(F1,F2) : F1 <= F2 <- +first(F1); +second(F2).

+!sort_forks(F1,F2) : F1 > F2 <- +first(F2); +second(F1).

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

+!acquireRes : first(F1) & second(F2)
  <- acquireFork(F1); acquireFork(F2).

+!releaseRes: first(F1) & second(F2)
  <- releaseFork(F1); releaseFork(F2).

+!think <- println("Thinking").
+!eat <- println("Eating").
```

REACTIVITY + PROACTIVITY

- Keeping a level of modularity & abstraction when integrating reactive and pro-active behavior
- Reactive philosopher example
  - *"..a philosopher must be able to react to alarms, so as to suspend ongoing activities and evacuate…"*
- Solution in Jason
  - extending the behaviour with a further plan:

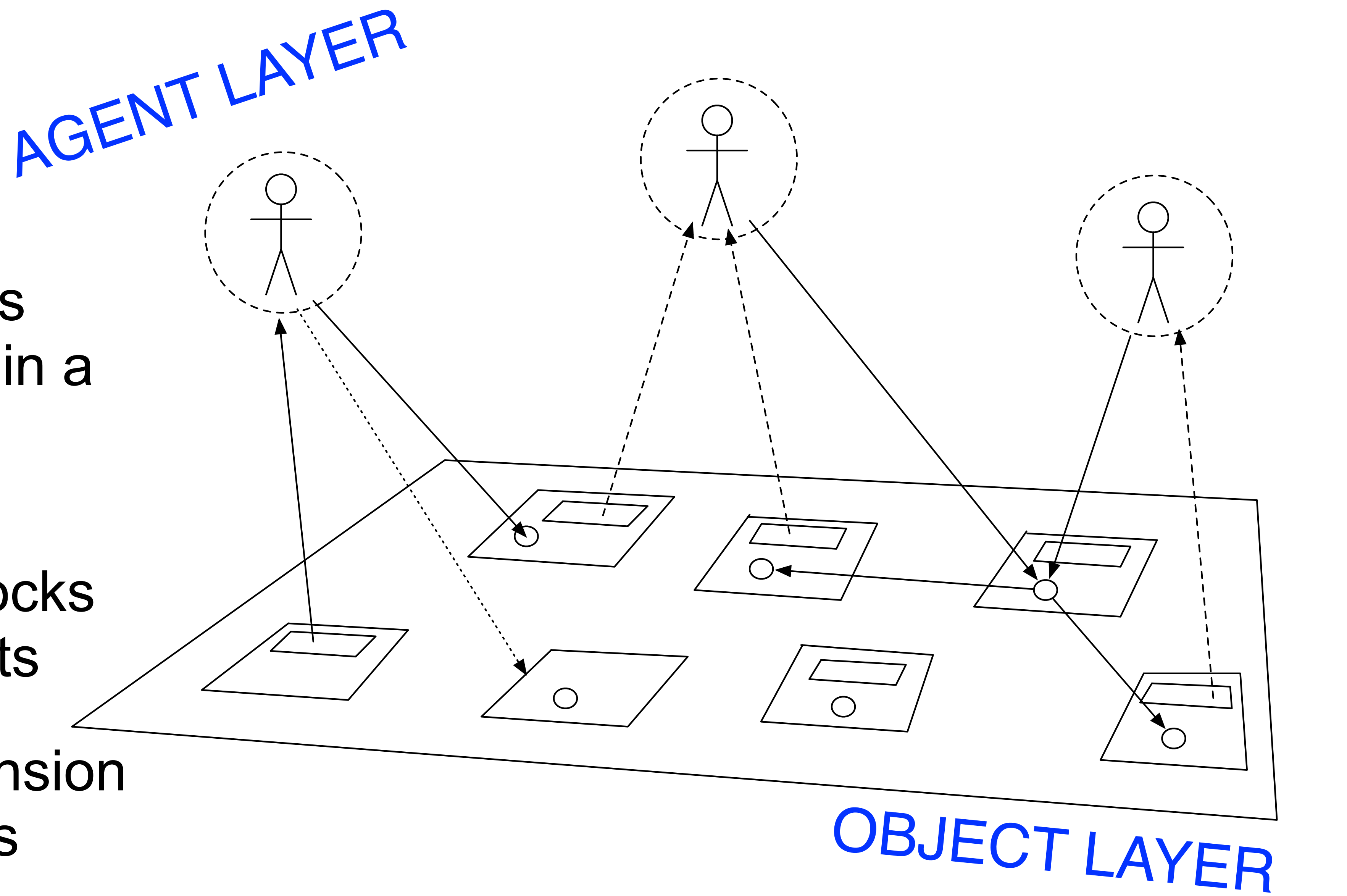
```
+alarm
  <- .drop_all_intentions; !evacuate.

+evacuate
  <- …
```

- *preserving modularity and abstraction*

ALOO

- Object-Oriented Concurrent Programming using agents
  - evolution of simpAL
- ALOO programs = agents + objects
  - agents ~ autonomous entities cooperating in a logically shared environment
  - objects ~ building blocks of agent environments
    - conservative extension of plain old objects
    - agent resources & coordination tools



ALOO CONTROL LOOP

Algorithm 4 ALOO control-loop

```
S ← S_0; PlanLib ← PlanLib_0 ; Ev ← {}
p ← SelectPlan(AssignedTask, PlanLib)
I ← {NewInt(p, AssignedTask)}
while I is not empty do
    currInt ← SelectIntention(I)
    ev ← FetchEvent(Ev, currInt)
    UpdateBel(currInt, ev)
    if ev is about a new sub-task t todo then
        p ← SelectPlan(t, PlanLib)
        PushPlan(currInt, p, t)
    else if ev is about a new task t todo then
        p ← SelectPlan(t, PlanLib)
        I ← I ∪ {NewInt(p, t)}
    end if
    a_l ← CollectActions(currInt, S, ev)
    for all a in a_l do
        ExecAction(a, currInt, S, I, PlanLib)
    end for
end while
```

SENSE

PLAN

ACT

Differently from Jason:

- the plan stage does not occur for every event
- events are considered in action selection
- more than one action can be selected and sequentially executed in a single cycle
- the loop is meant to terminate when all the tasks are done

DP IN ALOO

```
agent-script Philosopher  {
  public-tasks: DiningTask;
  Fork first, second;

  plan-for DiningTask {
    if (this-task.leftFork.id < this-task.rightFork.id){
      first = this-task.leftFork; second = this-task.rightFork
    } else {
      first = this-task.rightFork; second = this-task.leftFork
    };
    {
      always => {
        do ThinkingTask();
        do AcquiringForks();
        do EatingTask();
        do ReleasingForks()
      }
    }
  }
  plan-for ThinkingTask() { this-env.out.println("Thinking") }
  plan-for EatingTask() { this-env.out.println("Eating") }
  plan-for AcquireForks() { first.acquire() ; second.acquire() }
  plan-for ReleaseForks() { first.release() ; second.release() }
}
```

A .oop”

SUMMING UP

CONTROL ARCHITECTURE
ABSTRACTION & COMPLEXITY

threads	unconstrained control flow
receive-based actors	+ control flow encapsulation
event-loop actors	+ discipline and abstraction • reactive state machine
control-loop agents	+ abstraction • from messages to tasks + hierarchical structuring • plan-based

CONCLUDING REMARKS

- Limitations of this work and next steps
  - more examples, beyond Dining Philosophers
  - from a qualitative evaluation to a more objective/quantitative and methodical approach
  - programmer reasoning: *“simple programs running on complex loops”* vs. *“more complex programs running on simpler loops”*
  - mapping control loops on event loops
- Further investigations
  - impact of loops on composition, extensibility, reuse
  - impact of loops on performance
    - optimizations

AGERE! @ SPLASH 2014
Portland, Oregon

FROM ACTOR EVENT LOOP TO AGENT CONTROL LOOP: IMPACT ON PROGRAMMING

ALESSANDRO RICCI,
University of Bologna, Italy

+BEHAVIOURS: AKKA

```
class Hakker(name: String, left: ActorRef, right: ActorRef) extends Actor {
  ...
  def receive = {
    case Think => startThinking(5.seconds)
  }
  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 eating: Receive = { ... }
  ...
}
```

http://www.typesafe.com/activator/template/akka-sample-fsm-scala

+CONTINUATIONS: SALSA

```
behavior Philosopher {
  …
  Philosopher{Chopstick left, Chopstick right){ … }

  void eat(){ pickLeft() @ gotLeft(token); }

  boolean pickLeft(){ left <- get(self) @ currentContinuation; }

  void gotLeft(boolean leftOk){ … }

  void gotRight(boolean rightOk){
     join {
        standardOutput <- println ("eating...");
        left <- release();
        right <- release();
     } @ standardOutput <- println ("thinking...") @ eat();
  }
  …
}
```

C.Varela, "Programming with Actors", Chapter 9 of Programming Distributed Computing Systems. MIT Press