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

1: loop
2: \( msg \leftarrow \text{WAITFORMSG}() \)
3: \( h \leftarrow \text{SELECTMSGHANDLER}(msg) \)
4: \( args \leftarrow \text{GETMSGARGS}(msg) \)
5: \( \text{EXECUTEMSGHANDLER}(h, args) \)
6: 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

```java
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());
    }
}
```
IN THE PAPER

• Exploiting 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**

1: loop
2: \( msg \leftarrow \text{WAITFORMSG()} \)
3: \( o \leftarrow \text{LOCATEOBJECT}(msg) \)
4: \( m \leftarrow \text{GETMETHOD}(msg) \)
5: \( \text{args} \leftarrow \text{GETMETHODARGS}(msg) \)
6: \( \text{CALLMETHOD}(obj, m, \text{args}) \)
7: 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/AmbientTalk-CodeSnippets

- 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

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

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

1: $B \leftarrow B_0; \text{PlanLib} \leftarrow \text{PlanLib}_0; Ev \leftarrow \{\}; I \leftarrow \{\}$
2: loop
3: $\rho \leftarrow \text{SENESENV()}$
4: $\text{BELUPDATE}(\rho, B, Ev)$
5: if $Ev$ is not empty then
6: $ev \leftarrow \text{FETCHEVENT}(Ev)$
7: $p \leftarrow \text{SELECTPLAN}(ev, B, \text{PlanLib})$
8: if $ev$ is an env change or a new goal to achieve then
9: $I \leftarrow I \cup \{\text{NEWINT}(p, ev)\}$
10: else if $ev$ is a sub-goal to achieve then
11: $\text{PUSHPLAN}(\text{currInt}, p, ev)$
12: end if
13: end if
14: if $I$ is not empty then
15: $\text{currInt} \leftarrow \text{SELECTINTENTION}(I)$
16: $a \leftarrow \text{FETCHNEXTACTION}(\text{currInt})$
17: $\text{EXECACTION}(a, \text{currInt}, B, I, \text{PlanLib})$
18: end if
19: end loop

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
+!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:

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

+evacuate
  <- ...
```

• preserving modularity and abstraction
• 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

1: \( S \leftarrow S_0; \) PlanLib \( \leftarrow \) PlanLib0; Ev \( \leftarrow \) \{\}
2: \( p \leftarrow \) SELECTPLAN(AssignedTask, PlanLib)
3: \( I \leftarrow \) \{NEWINT\( (p, \) AssignedTask)\}
4: \textbf{while} \( I \) is not empty \textbf{do}
5: \( \text{currInt} \leftarrow \) SELECTINTENTION\( (I) \)
6: \( ev \leftarrow \) FETCHEVENT\( (Ev, \text{currInt}) \)
7: \( \) UPDATEBEL\( (\text{currInt}, ev) \)
8: \textbf{if} ev is about a new sub-task \( t \) todo \textbf{then}
9: \( p \leftarrow \) SELECTPLAN\( (t, \) PlanLib\)
10: \( \) PUSHPLAN\( (\text{currInt}, p, t) \)
11: \textbf{else if} ev is about a new task \( t \) todo \textbf{then}
12: \( p \leftarrow \) SELECTPLAN\( (t, \) PlanLib\)
13: \( I \leftarrow I \cup \) \{NEWINT\( (p, t) \}\}
14: \textbf{end if}
15: \( a_l \leftarrow \) COLLECTACTIONS\( (\text{currInt}, S, ev) \)
16: \textbf{for all} \( a \) in \( a_l \) do
17: \( \) EXECACTION\( (a, \text{currInt}, S, I, \) PlanLib\)
18: \textbf{end for}
19: \textbf{end while}

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
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() }
}
### SUMMING UP

<table>
<thead>
<tr>
<th>Control Architecture</th>
<th>Abstraction &amp; Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>threads</td>
<td>unconstrained control flow</td>
</tr>
<tr>
<td>receive-based actors</td>
<td>+ control flow encapsulation</td>
</tr>
</tbody>
</table>
| 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
FROM ACTOR EVENT LOOP TO AGENT CONTROL LOOP: IMPACT ON PROGRAMMING

ALESSANDRO RICCI,
University of Bologna, Italy
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
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();
        }
    …
}