PROGRAMMING ABSTRACTIONS FOR INTEGRATING AUTONOMOUS and REACTIVE BEHAVIORS: AN AGENT-ORIENTED APPROACH

ALESSANDRO RICCI, ANDREA SANTI
University of Bologna, Italy

AGERE! @ SPLASH 2012
Tucson, Arizona (US)
October 21, 2012

OUTLINE

OUTLINE

THE PROBLEM/OBJECTIVE

> sw components with autonomous + reactive behavior

> models, programming abstractions?

OUTLINE

THE PROBLEM/OBJECTIVE

> sw components with autonomous + reactive behavior

> models, programming abstractions?

DEALING IT WITH ACTORS: LIMITS

> reactivity principle pros & cons

OUTLINE

THE PROBLEM/OBJECTIVE
> sw components with autonomous + reactive behavior
> models, programming abstractions?

DEALING IT WITH ACTORS: LIMITS
> reactivity principle pros & cons

A SOLUTION BASED ON AGENTS
> control architecture for pro-activity + re-activity
> simpAL programming abstraction

INTEGRATING AUTONOMOUS & REACTIVE BEHAVIORS

INTEGRATING AUTONOMOUS & REACTIVE BEHAVIORS

Building autonomous components that are capable of
> ...promptly react to events...
> ...while pro-actively doing some (possibly infinite) set of actions
> keeping a clean & modular program design

INTEGRATING AUTONOMOUS & REACTIVE BEHAVIORS

Building autonomous components that are capable of
> ...promptly react to events...
> ...while pro-actively doing some (possibly infinite) set of actions
> keeping a clean & modular program design

Many examples in different domains
> web crawler reacting to user inputs
> distributed algorithms (mutex, election,..)
> control systems
> ...

Distributed mutual exclusion example [*]

Algorithm 10.2: Ricart-Agrawala algorithm

```
                integer myNum ← 0
                set of node IDs deferred ← empty set
                integer highestNum ← 0
                boolean requestCS ← false
```

Main

```
        loop forever
p1:         non-critical section
p2:         requestCS ← true
p3:         myNum ← highestNum + 1
p4:         for all other nodes N
p5:             send(request, N, myID, myNum)
p6:         await reply's from all other nodes
p7:         critical section
p8:         requestCS ← false
p9:         for all nodes N in deferred
p10:            remove N from deferred
p11:            send(reply, N, myID)
```

Receive

```
        integer source, requestedNum
        loop forever
p1:         receive(request, source, requestedNum)
p2:         highestNum ← max(highestNum, requestedNum)
p3:         if not requestCS or requestedNum ≪ myNum
p4:             send(reply, source, myID)
p5:         else add source to deferred
```

[*] Principle of Concurrent and Distributed Computing. Ben Ari. Addison Wesley

RELATED PROBLEMS...

> Integrating threads and events
> Events without inversion of control
> Asynchronous programming without spaghetti

....

TACKLING THE PROBLEM WITH ACTORS

TACKLING THE PROBLEM WITH ACTORS

ACTOR SEMANTICS

> based on pure reactivity principle

= an actor dormant until a message is received

TACKLING THE PROBLEM WITH ACTORS

ACTOR SEMANTICS

> based on pure reactivity principle
  = an actor dormant until a message is received

> macro-step semantics
= received msgs are processed one at a time in a *single atomic step* = all actions taken in response to the given message

TACKLING THE PROBLEM WITH ACTORS

ACTOR SEMANTICS

> based on pure reactivity principle
  = an actor dormant until a message is received

> macro-step semantics
  = received msgs are processed one at a time in a *single atomic step* = all actions taken in response to the given message

> analogous to event-loop frameworks (*run-to-completion* semantics)

TACKLING THE PROBLEM WITH ACTORS

ACTOR SEMANTICS

> based on pure reactivity principle
  = an actor dormant until a message is received

> macro-step semantics
  = received msgs are processed one at a time in a *single atomic step* = all actions taken in response to the given message

> analogous to event-loop frameworks (*run-to-completion* semantics)

=> *pro-activity must be implemented through reactivity*

A SIMPLE CASE STUDY

A SIMPLE CASE STUDY

- an actor doing a task *TaskT*, which can be decomposed in sub-tasks *ta, tb, tc*
- TaskT requires to promptly react to a message *react* that can be sent in any moment while doing TaskT
- upon receiving *react*, the actor must interrupt its current activity and do a sub-task *td*

A SIMPLE CASE STUDY

- an actor doing a task *TaskT*, which can be decomposed in sub-tasks *ta, tb, tc*
- TaskT requires to promptly react to a message *react* that can be sent in any moment while doing TaskT
- upon receiving *react*, the actor must interrupt its current activity and do a sub-task *td*

Studying the problem using:

> ActorFoundry

= Java framework with a rigorous actor semantics

> Erlang

= explicit receive (like Scala Actors, ...)

FIRST SOLUTION IN ACTOR-FOUNDRY

```
public class TestActor0 extends Actor {
  private int c = 0;

  @message
  public void doTaskT() {
    ta();
    tb();
    tc();
  }

  @message
  public void react() {
    call(stdout, "println","react! "+c);
  }

  private void ta(){ c = c + 1; }
  private void tb(){ c = c + 1; }
  private void tc(){ c = c + 1; }
}
```

FIRST SOLUTION IN ACTOR-FOUNDRY

```
public class TestActor0 extends Actor {
  private int c = 0;

  @message
  public void doTaskT() {
    ta();
    tb();
    tc();
  }

  @message
  public void react() {
    call(stdout, "println","react! "+c);
  }

  private void ta(){ c = c + 1; }
  private void tb(){ c = c + 1; }
  private void tc(){ c = c + 1; }
}
```

PROBLEM

> reactivity *shadowed*

= unique msg printed: react! 3

```
public class TestActor1 extends Actor {
  private int c = 0;

  @message
  public void doTaskT() {
    send(self(), "doingTa");
  }

  @message
  public void doingTa() {
    send(self(), "doingTb");
    ta();
  }

  @message
  public void doingTb() {
    send(self(), "doingTc");
    tb();
  }
  ...
  @message
  public void react() {
    call(stdout, "println","react! "+c);
  }
}
```

PARTIALLY IMPROVED SOLUTION

PARTIALLY IMPROVED SOLUTION

```
public class TestActor1 extends Actor {
  private int c = 0;

  @message
  public void doTaskT() {
    send(self(), "doingTa");
  }

  @message
  public void doingTa() {
    send(self(), "doingTb");
    ta();
  }

  @message
  public void doingTb() {
    send(self(), "doingTc");
    tb();
  }
  ...
  @message
  public void react() {
    call(stdout, "println","react! "+c);
  }
}
```

More reactive but *tricky design*

> to recover reactivity => need to break into multiple msg handlers + self-sending

> decomposition driven by the granularity of reactivity (vs. task complexity)

ERLANG SOLUTION

```
test_actor() ->
    self() ! doTaskT, loop(0).
loop(C) ->
    receive
      doTaskT ->
        C1 = ta(C),
        self() ! doingTb,
        loop(C1);
      doingTb ->
        C1 = tb(C),
        self() ! doingTc,
        loop(C1);
      doingTc ->
        C1 = tc(C),
        loop(C1);
      react ->
        io:format("react! ~w~n", [C]),
        loop(C)
    end.

ta(C) -> C+1.
...
```

ERLANG SOLUTION

```
test_actor() ->
    self() ! doTaskT, loop(0).
loop(C) ->
    receive
      doTaskT ->
        C1 = ta(C),
        self() ! doingTb,
        loop(C1);
      doingTb ->
        C1 = tb(C),
        self() ! doingTc,
        loop(C1);
      doingTc ->
        C1 = tc(C),
        loop(C1);
      react ->
        io:format("react! ~w~n", [C]),
        loop(C)
    end.

ta(C) -> C+1.
...
```

Solution with explicit receive

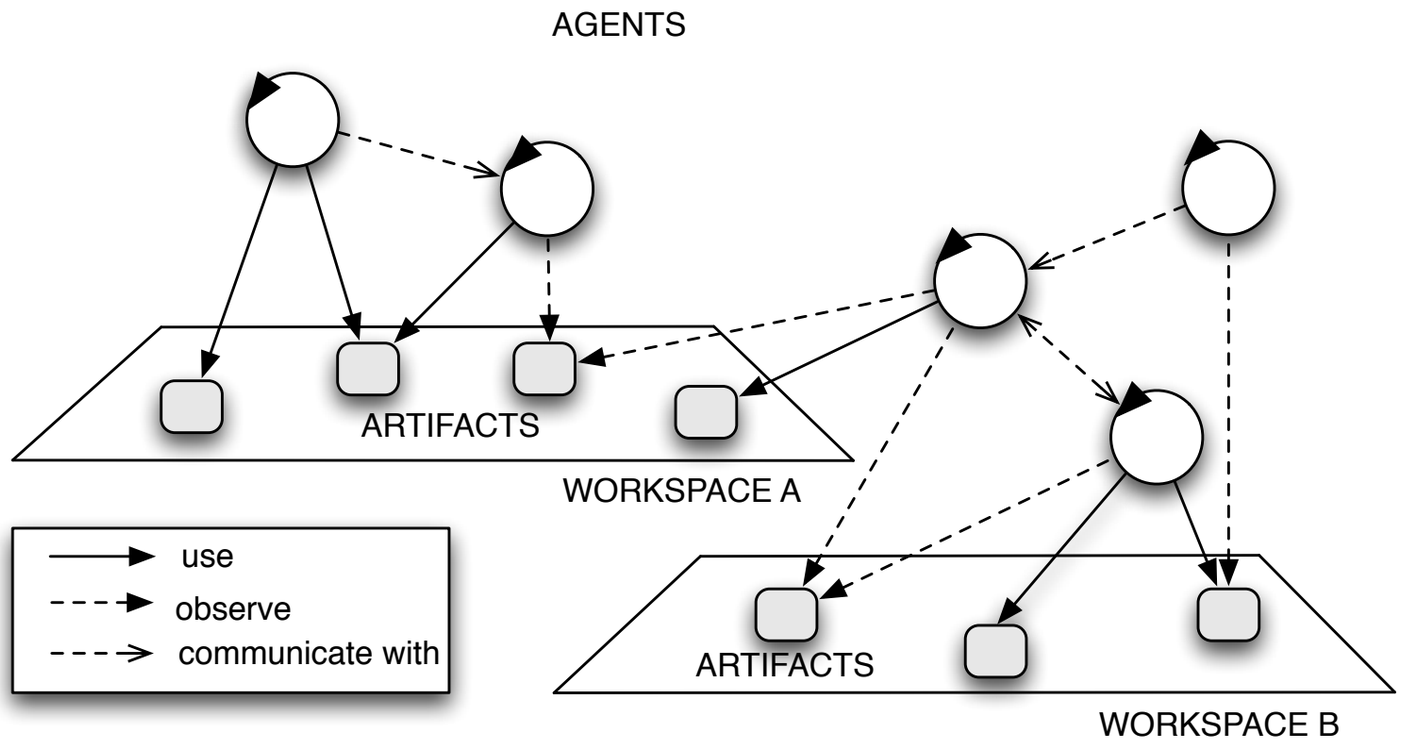
> same “artificial” decomposition & self-sending pollution...

TACKLING THE PROBLEM WITH AGENT-ORIENTED ABSTRACTIONS

TACKLING THE PROBLEM WITH AGENT-ORIENTED ABSTRACTIONS

Agent-Oriented Programming in simpAL

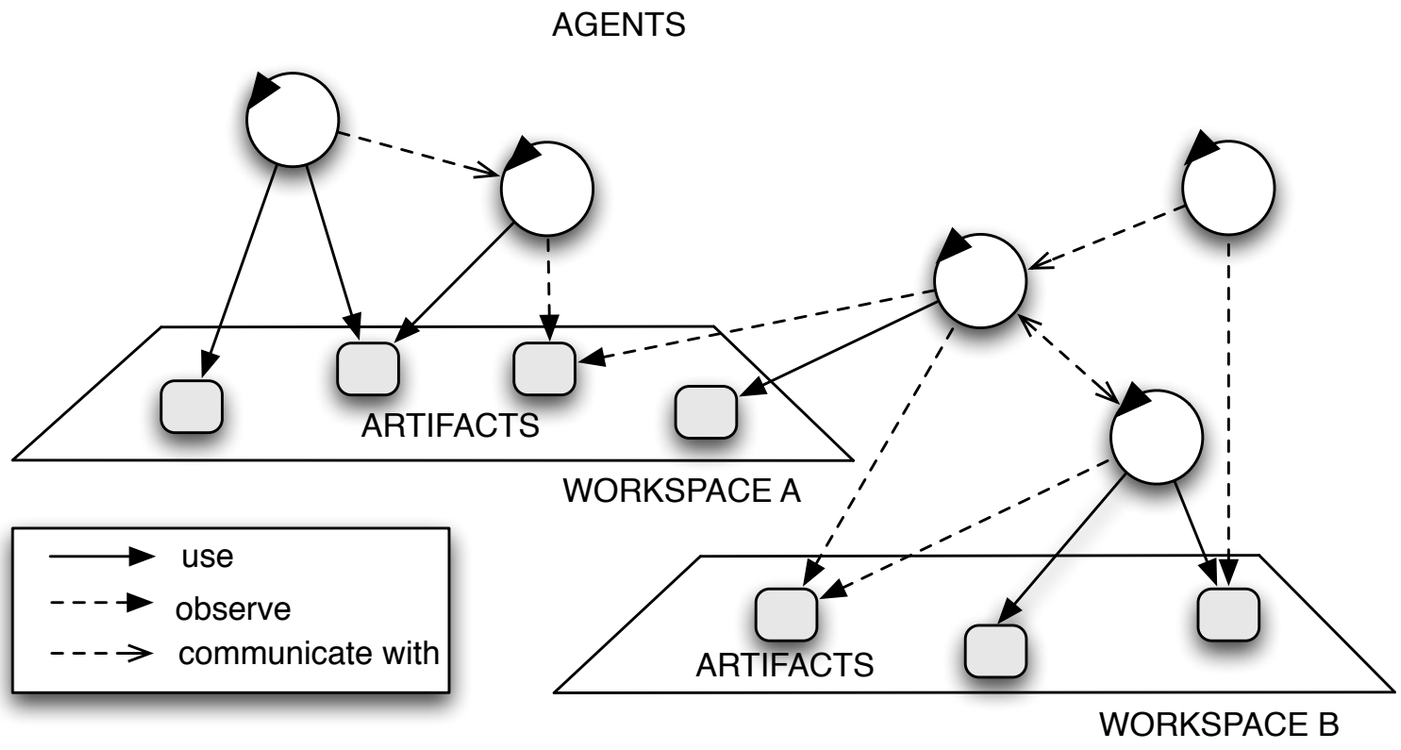
> programs as organizations of agents working in a shared environment



TACKLING THE PROBLEM WITH AGENT-ORIENTED ABSTRACTIONS

Agent-Oriented Programming in simpAL

> programs as organizations of agents working in a shared environment



Agent abstraction

> modeling autonomous components capable of achieving *tasks*

> conceptual extension/specialization of actors

PRO-ACTIVITY PRINCIPLE

PRO-ACTIVITY PRINCIPLE

Key feature of the agent model for our problem

Actors => reactivity principle
Agents => *pro-activity principle*

PRO-ACTIVITY PRINCIPLE

Key feature of the agent model for our problem

Actors => reactivity principle
Agents => *pro-activity principle*

> tasks as first-class concepts
> an agent acts because of a task to do *(state condition)*
> not (only) because of a message to handle *(event condition)*

EXTENSION OF THE CONTROL ARCHITECTURE

EXTENSION OF THE CONTROL ARCHITECTURE

Actors => event loop

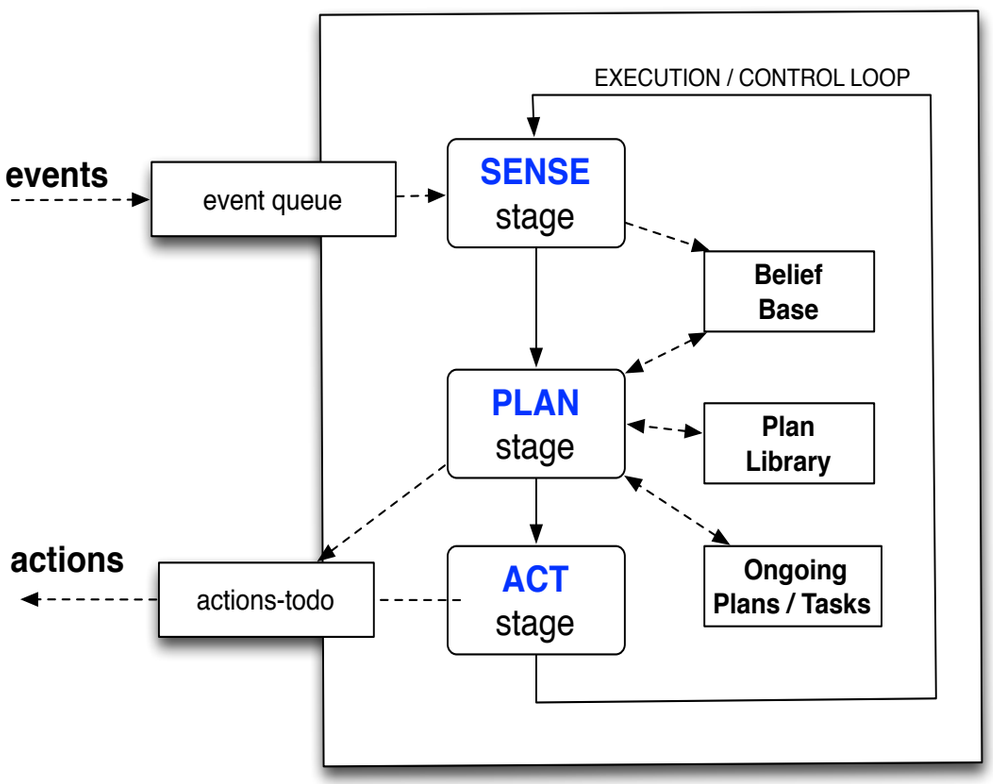
1: **while** $true$ **do**
2: $msg \leftarrow$ PICKMSGFROMMSGQUEUE()
3: $method \leftarrow$ SELECTHANDLER(msg)
4: EXECUTE($method$)
5: **end while**

EXTENSION OF THE CONTROL ARCHITECTURE

Agents => *control loop*

> agent execution cycle

conceptually never blocked



BDI-inspired architecture

```
1:  while true do
2:                                                        ▷ SENSE stage
3:      if external events queue not empty then
4:          ev ← PickExtEvent()
5:          UpdateAgentState(ev)
6:      end if
7:                                                        ▷ PLAN stage
8:      if new tasks todo then
9:          for each new task to-do task do
10:             plan ← SelectPlan(task, belBase, planLib)
11:             CreateNewIntention(plan, task)
12:         end for
13:     end if
14:     actList ← []
15:     for each ongoing intention i do
16:         if TaskFulfilled(i) then
17:             DropIntention(i)
18:         end if
19:         actList ← actList + SelectActions(i, belBase)
20:     end for
21:                                                       ▷ ACT stage
22:     for each action act in actList do
23:         Execute(act)
24:     end for
25: end while
```

TASKS AND PLANS AS PROGRAMMING ABSTRACTIONS in simpAL

TASKS

> description of a job to be done
> task types grouped in *roles*
> roles = agent *types*

PLANS

> specify how to accomplish tasks
> collected in *scripts*
> loaded dynamically by agents into their plan library

TASKS AND PLANS AS PROGRAMMING ABSTRACTIONS in simpAL

TASKS

> descripti
> task type
> roles = a

PLANS

> specify h
> collecte
> loaded d

o their plan library

```
role Thermostat {
  task KeepTemperature {
    input-params {
      targetTemp: double
    }
    output-params {
      ...
    }
    understands {
      newThreshold:double
    }
  }

  task AchieveTemperature {
    ...
  }
  ...
}
```

TASKS AND PLANS AS PROGRAMMING ABSTRACTIONS in simpAL

TASKS

> description of a job to be done
> task types grouped in *roles*
> roles = agent *types*

PLANS

> specify how to accomplish tasks
> collected in *scripts*
> loaded dynamically by agents into their plan library

TASKS AND PLANS AS PROGRAMMING ABSTRACTIONS in simpAL

TASKS

> description of
> task types grou
> roles = agent *t*

PLANS

> specify how to
> collected in *scr*
> loaded dynami ary

```
agent-script ACMEThermostat
                    implements Thermostat {

  energyLevel: double;
  ...

  plan-for KeepTemperature
          context: energyLevel > 10{
    ...
  }

  plan-for KeepTemperature
          context: energyLevel <= 10{
    ...
  }

  plan-for AchieveTemperature { ... }

}
```

TASKS AND PLANS AS PROGRAMMING ABSTRACTIONS in simpAL

TASKS

> description of a job to be done
> task types grouped in *roles*
> roles = agent *types*

PLANS

> specify how to accomplish tasks
> collected in *scripts*
> loaded dynamically by agents into their plan library

PLAN MODEL IN simpAL

PLAN BODY

PLAN MODEL IN simpAL

PLAN BODY

> set of action rules specify how to act and react

each rule: *when* to do *what* (ECA-like):

```
("when"|"every-time")[Ev][":" Cond] => Act [#lbl]
```

PLAN MODEL IN simpAL

PLAN BODY

> set of action rules specify how to act and react
each rule: *when* to do *what* (ECA-like):

```
("when"|"every-time")[Ev][":" Cond] => Act [#lbl]
```

> can describe any arbitrary combination of sequence of *actions* with *reactions* = actions to be taken if/when/every-time some event/condition hold

PLAN MODEL IN simpAL

PLAN BODY

> set of action rules specify how to act and react
each rule: *when* to do *what* (ECA-like):

```
("when"|"every-time")[Ev][":" Cond] => Act [#lbl]
```

> can describe any arbitrary combination of sequence of *actions* with *reactions* = actions to be taken if/when/every-time some event/condition hold

> can be organized in possibly nested *action rule blocks*

PLAN MODEL IN simpAL

PLAN BODY

> set of action rules specify how to act and react
each rule: *when* to do *what* (ECA-like):

```
("when"|"every-time")[Ev][":" Cond] => Act [#lbl]
```

> can describe any arbitrary combination of sequence of *actions* with *reactions* = actions to be taken if/when/every-time some event/condition hold

> can be organized in possibly nested *action rule blocks*

> syntactic sugar for representing more frequent control pattern (e.g. sequence)

ACTION RULE EXEC SEMANTICS

> in PLAN stage, every action rule *ar* such that the *when* condition is satisfied is collected
> in the ACT stage actions of triggered rules are executed
> triggered actions can be action rule block
(=> pushed on the rule block stack)
(=> ~interrupting current action rule block)

THE ABSTRACT EXAMPLE

```
role RoleR {
  task TaskT {
    input-params {}
    output-params {}
    understands {
      react: boolean
    }
  }
}
```

```
agent-script TestScript implements RoleR {
  c: int = 0;

 plan-for TaskT completed-when: is-done tc {
    do-task new-task Ta();
    do-task new-task Tb();
    do-task new-task Tc() #tc

    when told this-task.react
                => using: console@main {
      println(msg: "react! "+c)
    }
  }

  plan-for Ta { c = c + 1  }
  plan-for Tb { c = c + 1  }
  plan-for Tc { c = c + 1  }

  /* private tasks */
  task Ta{}
  task Tb{}
  task Tc{}
}
```

ASYNCHRONOUS PROGRAMMING

```
agent-script MyClient implements Client {
  inputGUI: UserView
  ...
  plan-for UseServices  using: inputGUI
                        completed-when: stopped {
    always =>
           completed-when: (is-done reqC || stopped)
           using: serviceA, serviceB {

       doReqA() #reqA
       doReqB() #reqB
       when is-done reqA && is-done reqB
                                  => using:serviceC {
         doReqC() #reqC
       }
    }
  }
}
```

EVENT PROGRAMMING WITHOUT INVERSION OF CONTROL

```
usage-interface CounterUI {
  obs-prop count: int;
  operation inc();
}

artifact Counter implements CounterUI {
  init() { count = 0; }
  operation inc () { count = count + 1; }
}
```

```
role Observer {
  task Observing {
    input-params {
     sharedCounter: CounterUI;
}}}

agent-script ObserverScript implements Observer {
  plan-for Observing using: console@main {
    println(msg: "start observing...");
    using: this-task.sharedCounter {
      every-time changed count => {
        println(msg: "new count perceived! ")
}}}}
```

MANAGING TASKS AS FIRST-CLASS ENTITIES

```
agent-script TestScript implements RoleR {
  ...
  plan-for TaskT completed-when: is-done tc || is-done td {
     do-task new-task Ta();
     do-task new-task Tb();
     do-task new-task Tc() #tc

     when told this-task.react => using: console@main {
       drop-all-tasks ;
       forget-old-plans ;
       println(msg: "react! "+c) ;
       do-task new-task Td() #td
     }
  }

  plan-for Td using: console@main {...}
  ...
}
```

MANAGING TASKS AS FIRST-CLASS ENTITIES

```
plan-for TaskT completed-when: is-done tc {
    ta: Ta = new-task Ta();
    do-task ta;
    tb: Tb = new-task Tb();
    do-task tb;
    ...
    when told this-task.react :
              is-ongoing ta || is-ongoing tb =>
        using: console@main
          completed-when: is-done te || is-done newTc {
      te: Te; newTc: Tc;
      forget-old-plans ;
      atomically: {
        println(msg: "react! "+c) ;
        if (is-ongoing ta) {
          drop-task ta ;
          newTc = new-task Tc() ;
          assign-task newTc ;
        } else-if (is-ongoing tb) {
          te = new-task Te( prev: tb) ;
          assign-task te
}}}}
```

A REAL-WORLD EXAMPLE

Reactive File Searcher
> collecting all files with a size greater than X
> users can change the file dimension dynamically
> users can stop the process

(in the paper)

FIRST PERFORMANCE EVALUATION

FIRST PERFORMANCE EVALUATION

Two main critical points about performance in simpAL

> uncoupling between logical concurrency (agent-oriented layer) & physical concurrency (threads)

> agent control loop & plan model based on ECA-like action rules
- conceptually the agent cycle is never blocked

FIRST PERFORMANCE EVALUATION

Two main critical points about performance in simpAL

> uncoupling between logical concurrency (agent-oriented layer) & physical concurrency (threads)

> agent control loop & plan model based on ECA-like action rules
- conceptually the agent cycle is never blocked

First optimizations

> don't cycle if not needed

FIRST PERFORMANCE EVALUATION

Two main critical points about performance in simpAL

> uncoupling between logical concurrency (agent-oriented layer) & physical concurrency (threads)

> agent control loop & plan model based on ECA-like action rules
- conceptually the agent cycle is never blocked

First optimizations

> don't cycle if not needed

First tests are encouraging

- test reported in the paper with a long-running autonomous behavior that must react a number of times to messages/events

FIRST PERFORMANCE EVALUATION

Number of Iterations	Erlang	ActorFoundry	Jason	simpAL
10000 Iterations	1.52 s	8,92 s	12,56 s	1,58 s
10000 Iterations (no print)	0.04 s	0.06 s	6,30 s	0.89 s
25000 Iterations	8.82 s	56,94 s	69.57 s	4,52 s
25000 Iterations (no print)	0.06 s	0.14 s	50.69 s	2,29 s
50000 Iterations	35,34 s	218,64 s	281,16 s	9,15 s
50000 Iterations (no print)	0.11 s	0.31 s	252,95 s	4,55 s

First tests are encouraging

- test reported in the paper with a long-running autonomous behavior that must react a number of times to messages/events

RELATED WORK /1

RELATED WORK /1

Works integrating threads and event-driven programming

- Scala actors & "Unifying threads and events"
- Kilim - using CPS to integrate ultra-light weight thread and events
- Non-actor ones: Task Java, AC, F#, GHC Haskell...
  event-driven programming without inversion of control
- Behavioral Programming - b-threads abstraction directly encapsulating requirement of a reactive system

RELATED WORK /1

Works integrating threads and event-driven programming

- Scala actors & "Unifying threads and events"
- Kilim - using CPS to integrate ultra-light weight thread and events
- Non-actor ones: Task Java, AC, F#, GHC Haskell... event-driven programming without inversion of control
- Behavioral Programming - b-threads abstraction directly encapsulating requirement of a reactive system

=> simpAL:

- allows for reacting while acting
- supporting structured and nested blocks with autonomous+reactive behaviors

RELATED WORK /2

RELATED WORK /2

Existing BDI agent-programming languages DAI in literature

- several examples: Jason/AgentSpeak(L), JACK, 2APL, GOAL, AgentFactory...

RELATED WORK /2

Existing BDI agent-programming languages DAI in literature

- several examples: Jason/AgentSpeak(L), JACK, 2APL, GOAL, AgentFactory...

=> simpAL:

- very similar control architecture (in particular with AgentSpeak/Jason)
- different plan model to improve modularization & encapsulation
- different handling of multiple tasks execution
- not targeted to (D)AI but to concurrent & distributed programming

ONGOING WORK

ONGOING WORK

Formalization of the agent control loop & simpAL

> rigorously specifying the semantics

> investigating properties

ONGOING WORK

Formalization of the agent control loop & simpAL
> rigorously specifying the semantics
> investigating properties

More performance analysis and evaluation
> optimizations

ONGOING WORK

Formalization of the agent control loop & simpAL
> rigorously specifying the semantics
> investigating properties

More performance analysis and evaluation
> optimizations

About simpAL
> cooperative tasks
> sub-typing
> reuse at the agent/artifact level
> platform improvement
(compiler, runtime, IDE, libraries)

THANKS