

# Multi-Agent Programming

## – Programming Autonomous Agents –

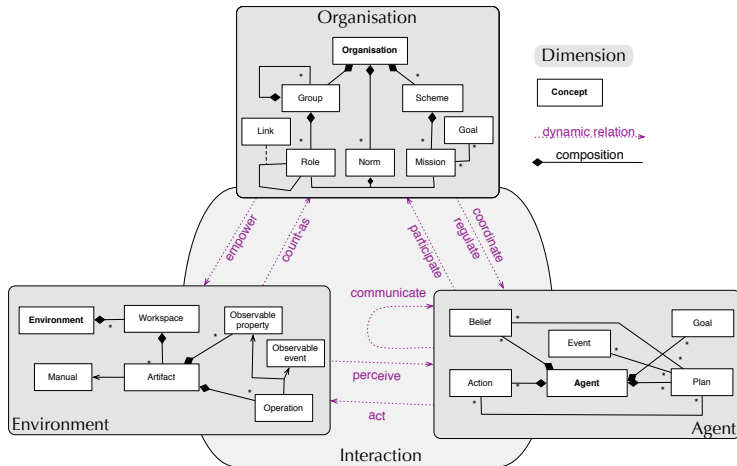
O. Boissier

Univ. Lyon, IMT Mines Saint-Etienne, LaHC UMR CNRS 5516, France

CPS2 M1 – Fall 2020



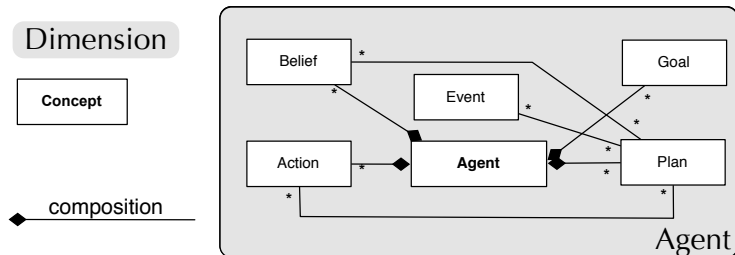
# JaCaMo meta-model



Simplified view on JaCaMo meta-model [Boissier et al., 2020, Boissier et al., 2011]

A seamless integration of three dimensions based on **Jason** [Bordini et al., 2007],  
**Cartago** [Ricci et al., 2009], **Moise** [Hübner et al., 2009] meta-models

# Agent dimension



Simplified Conceptual View (Jason meta-model [Bordini et al., 2007]):

Simple Agent Program:

```
happy(bob). // initial belief
!say(hello). // initial goal
/* Plans */
+!say(X) : happy(bob) <- .print(X).
// ...
```

example bob.asl

```
+happy(A) <- !say(hello(A)).
+!say(A) : not today(friday) <- .print(X); !say(X).
+!say(X) : today(friday) <- .print("stop").
-happy(A) : .my_name(A) <- .drop_intention(say(_)).
```

example carl.asl

# Agent in JaCaMo: *Jason*

The foundational language for *Jason* is AgentSpeak

- ▶ Originally proposed by Rao [Rao, 1996]
- ▶ Programming language for BDI agents
- ▶ Elegant notation, based on **logic programming**
- ▶ Inspired by PRS [Georgeff and Lansky, 1987], dMARS [d'Inverno et al., 1997], and BDI Logics [Rao et al., 1995]
- ▶ Abstract programming language aimed at theoretical results

# Jason

A practical implementation of a variant of AgentSpeak

- ▶ *Jason* implements the **operational semantics** of a variant of AgentSpeak
- ▶ Has various extensions aimed at a more **practical** programming language (e.g. definition of the MAS, communication, ...)
- ▶ Highly customised to simplify **extension** and **experimentation**
- ▶ Developed by Jomi F. Hübner, Rafael H. Bordini, and others

# Outline

Agent Abstractions

Agent Dynamics

Other language features

Integrating **A** & **A** dimensions

Agent Management Infrastructure in JaCaMo

Comparison with other paradigms

Conclusions and wrap-up

# Main Language Constructs

**Beliefs:** represent the information available to an agent (e.g. about the environment or other agents)

**Goals:** represent states of affairs the agent wants to bring about

**Plans:** are recipes for action, representing the agent's know-how  
**Actions** can be internal, external, communicative or organisational ones

**Events:** happen as consequence to changes in the agent's beliefs or goals

**Intentions:** plans instantiated to achieve some goal

**Note:** identifiers starting in upper case denote variables

# Main Language Constructs and Runtime Structures

**Beliefs:** represent the information available to an agent (e.g. about the environment or other agents)

**Goals:** represent states of affairs the agent wants to bring about

**Plans:** are recipes for action, representing the agent's know-how  
**Actions** can be internal, external, communicative or organisational ones

Runtime structures:

**Events:** happen as consequence to changes in the agent's beliefs or goals

**Intentions:** plans instantiated to achieve some goal

**Note:** identifiers starting in upper case denote variables



## (BDI & Jason) Hello World – agent bob

```
happy(bob) .                // B
!say(hello) .               // D

+!say(X) : happy(bob) <- .print(X).
```

beliefs: prolog like (First Order Logic)

## (BDI & Jason) Hello World – agent bob

```
happy(bob) .                // B
!say(hello) .               // D

+!say(X) : happy(bob) <- .print(X).
```

beliefs: prolog like (First Order Logic)

desires: prolog like, with ! prefix

## (BDI & Jason) Hello World – agent bob

```
happy(bob) .                // B
!say(hello) .               // D

+!say(X) : happy(bob) <- .print(X).
```

beliefs: prolog like (First Order Logic)

desires: prolog like, with ! prefix

plans:

- ▶ define when a desire becomes an intention  $\leadsto$  **deliberate**
- ▶ how it is satisfied
- ▶ are used for practical reasoning  $\leadsto$  **means-end**

# (BDI & Jason) Hello World – agent bob

desires from perception – options

```
+happy(bob) <- !say(hello).
```

```
+!say(X) : not today(monday) <- .print(X).
```

# (BDI & Jason) Hello World – agent bob

source of beliefs

```
+happy(bob) [source(A)]  
  : someone_who_knows_me_very_well(A)  
  <- !say(hello).  
  
+!say(X) : not today(monday) <- .print(X).
```

# (BDI & Jason) Hello World – agent bob

plan selection

```
+happy(H) [source(A)]  
  : sincere(A) & .my_name(H)  
  <- !say(hello).
```

```
+happy(H)  
  : not .my_name(H)  
  <- !say(i_envy(H)).
```

```
+!say(X) : not today(monday) <- .print(X).
```

# (BDI & Jason) Hello World – agent bob

intention revision

```
+happy(H) [source(A)]  
  :   sincere(A) & .my_name(H)  
  <- !say(hello).
```

```
+happy(H)  
  :   not .my_name(H)  
  <- !say(i_envy(H)).
```

```
+!say(X) : not today(monday) <- .print(X); !say(X).
```

```
-happy(H)  
  :   .my_name(H)  
  <- .drop_intention(say(hello)).
```

# (BDI & Jason) Hello World – agent bob

intention revision

```
+happy(H) [source(A)]  
  :   sincere(A) & .my_name(H)  
  <- !say(hello).
```

```
+happy(H)  
  :   not .my_name(H)  
  <- !say(i_envy(H)).
```

```
+!say(X) : not today(monday) <- .print(X); !say(X).
```

```
-happy(H)  
  :   .my_name(H)  
  <- .drop_intention(say(hello)).
```



# (BDI & Jason) Hello World – agent bob

intention revision / Features

- ▶ we can have several intentions based on the same plans
- ~> running concurrently
- ▶ long term goal running
- ~> reaction meanwhile!

# Beliefs representation

## Agent Abstractions

### Syntax

Beliefs are represented by annotated literals of first order logic

```
functor(term1, ..., termn)[annot1, ..., annotm]
```

### Example (belief base of agent Tom)

```
red(box1)[source(percept)].  
friend(bob,alice)[source(bob)].  
lier(alice)[source(self),source(bob)].  
~lier(bob)[source(self)].
```

# Goals representation

## Agent Abstractions

### Syntax

Goals are represented as beliefs with a prefix:

- ▶ **!** to denote achievement goal (goal **to do**)
- ▶ **?** to denote test goal (goal **to know**)

### Example (Initial goal of agent Tom)

**!**write(book).

# Plans representation

## Agent Abstractions

### Syntax

An AgentSpeak plan has the following general structure:

`triggering_event : context <- body.`

where:

- ▶ `triggering_event`: events that the plan is meant to handle
- ▶ `context`: situations in which the plan can be used
- ▶ `body`: course of action to be used to handle the event if the context is believed to be true at the time a plan is being chosen to handle the event

# Plans representation – Triggering events

## Agent Abstractions

- ▶ Events happen as consequence to changes in the agent's beliefs or goals
- ▶ An agent reacts to events by executing **plans**

### Syntax

- ▶ belief addition: **+b**
- ▶ belief deletion: **-b**
- ▶ achievement-goal addition: **+!g**
- ▶ achievement-goal deletion: **-!g**
- ▶ test-goal addition: **+?g**
- ▶ test-goal deletion): **-?g**

# Plans representation – Context

## Agent Abstractions

Context is a boolean expression with the following operators:

### Syntax

#### ► Boolean operators

**&** (and)

| (or)

**not** (not)

= (unification)

>, >= (relational)

<, <= (relational)

== (equals)

\ == (different)

#### ► Arithmetic operators

+

- (subtraction)

\*

/ (divide)

**div** (divide – integer)

**mod** (remainder)

**\*\*** (power)

# Plans representation – Body

## Agent Abstractions

A plan body may contain:

- ▶ Belief operators
  - + (new belief)
  - (dispose belief)
  - + (update belief)
- ▶ Goal operators
  - ! (new achievement sub-goal)
  - ? (new test sub-goal)
  - !! (new achievement goal)
- ▶ External actions defined from artifact operations (see course on **Agent Working Environment**)
- ▶ Internal actions
  - ▶ Unlike actions, internal actions do not change the environment
  - ▶ Encapsulate code to be executed as part of the agent reasoning cycle
  - ▶ Internal actions can be used for invoking legacy code
- ▶ Constraints

# Internal Actions

## Agent Abstractions

- ▶ Internal actions can be defined by the user in Java  
`libname.action_name(...)`
- ▶ Standard (pre-defined) internal actions in standard library (no library name):
  - ▶ `.print(term1, term2, ...)`
  - ▶ `.union(list1, list2, list3)`
  - ▶ `.my_name(var)`
  - ▶ `.send(ag, perf, literal)`
  - ▶ `.intend(literal)`
  - ▶ `.drop_intention(literal)`
- ▶ Many others available for: printing, sorting, list/string operations, manipulating the beliefs/annotations/plan library, creating agents, waiting/generating events, etc.



# Plans representation

## Agent Abstractions

### Example

```
+rain :  time_to_leave(T) & clock.now(H) & H >= T
  <- !g1;           // new sub-goal
      !!g2;          // new goal
      ?b(X);         // new test goal
      +b1(T-H);      // add mental note
      -b2(T-H);      // remove mental note
      -+b3(T*H);     // update mental note
      jia.get(X);    // internal action
      X > 10;        // constraint to carry on
      close(door);   // external action
      !g3[hard_deadline(3000)]. // goal with deadline
```

# Plans representation

## Agent Abstractions

### Example

```
+green_patch(Rock) [source(percept)]
:  not battery_charge(low)
<- ?location(Rock,Coordinates);
    !at(Coordinates);
    !examine(Rock).

+!at(Coords)
:  not at(Coords) & safe_path(Coords)
<- move_towards(Coords);
    !at(Coords).

+!at(Coords)
:  not at(Coords) & not safe_path(Coords)
<- ...

+!at(Coords) :  at(Coords).
```

# Outline

Agent Abstractions

Agent Dynamics

Other language features

Integrating **A** & **A** dimensions

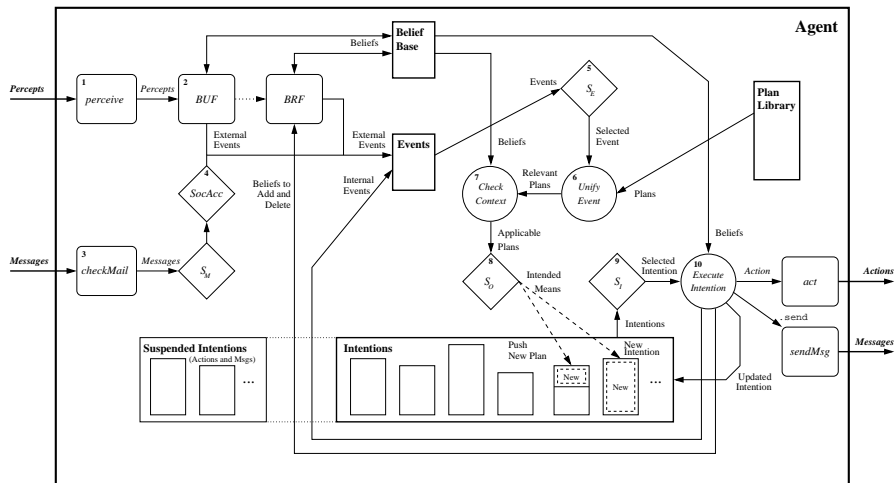
Agent Management Infrastructure in JaCaMo

Comparison with other paradigms

Conclusions and wrap-up

# Agent dynamics

## Agent Dynamics

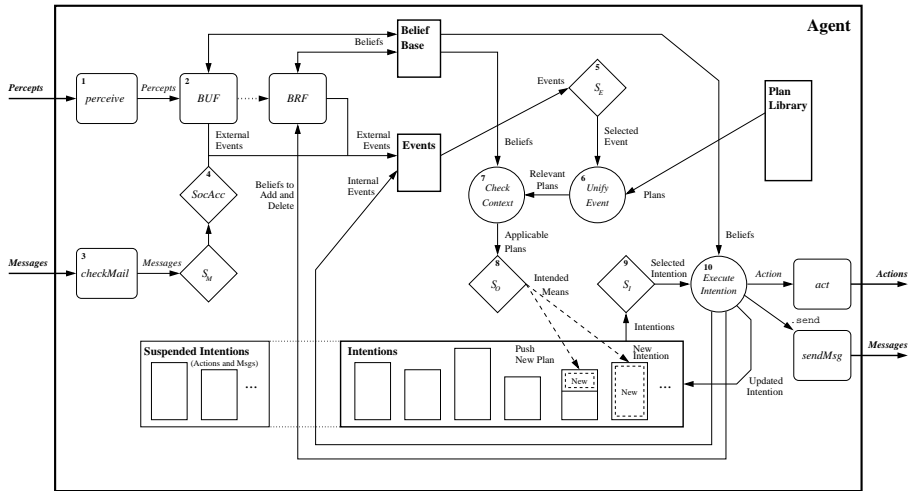


# Basic Reasoning cycle

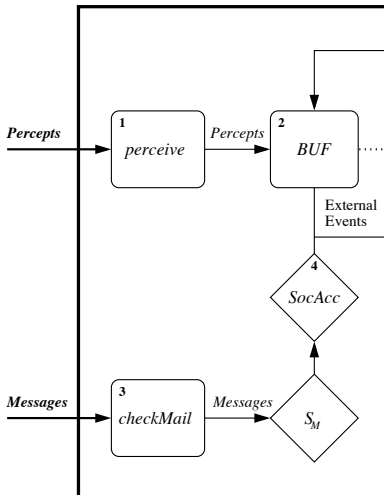
runtime interpreter

- ▶ perceive the environment and update belief base
- ▶ process new messages
- ▶ select event
- ▶ select **relevant** plans
- ▶ select **applicable** plans
- ▶ create/update intention
- ▶ select intention to execute
- ▶ execute one step of the selected intention

# Basic Reasoning Cycle

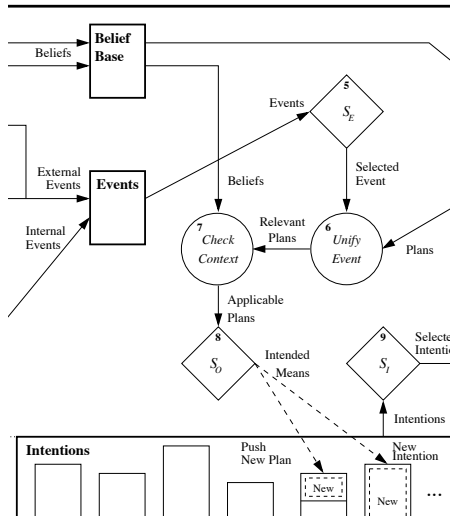


## Basic Reasoning Cycle



- ▶ machine perception
- ▶ belief revision
- ▶ knowledge representation
- ▶ communication, argumentation
- ▶ trust
- ▶ social power

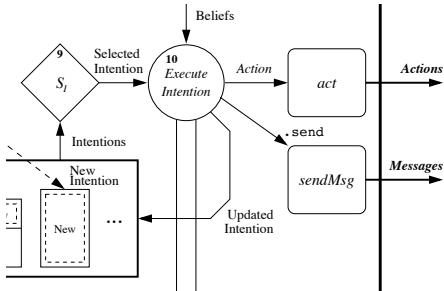
# Basic Reasoning Cycle



- ▶ planning
- ▶ reasoning
- ▶ decision theoretic techniques
- ▶ learning (reinforcement)



# Basic Reasoning Cycle



- ▶ intention reconsideration
- ▶ scheduling
- ▶ action theories

# Beliefs dynamics

## Agent Dynamics

### Internal reasoning

The **plan operators** + and - can be used to add and remove beliefs annotated with `source(self)` (**mental notes**)

```
+lier(alice); // adds lier(alice)[source(self)]  
-lier(john); // removes lier(john)[source(self)]
```

### Perception (from the environment)

Beliefs are automatically updated accordingly to the perception of the agent (annotated with `source(percept)`)

# Beliefs dynamics

## Agent Dynamics

### Communication (from other agents)

When an agent receives a **tell** (resp. **untell**) message, the content is a new belief (annotated with the sender of the message) (resp. belief corresponding to the content is deleted)

```
.send(tom,tell,lier(alice)); // sent by bob
// adds lier(alice)[source(bob)] in Tom's Belief Base
...
.send(tom,untell,lier(alice)); // sent by bob
// removes lier(alice)[source(bob)] from Tom's Belief Base
```

# Goals dynamics

## Agent Dynamics

### Internal reasoning

The **plan operators** **!**, **!!** and **?** are used to add a new goal (annotated with **source(self)**)

...

```
// adds new achievement goal !write(book)[source(self)]
```

```
!write(book);
```

```
// adds new test goal ?publisher(P)[source(self)]
```

```
?publisher(P);
```

...

# Goals dynamics

## Agent Dynamics

### Communication of achievement goal

When an agent receives an **achieve** message, the content is a new achievement goal (annotated with the sender of the message)

```
.send(tom,achieve,write(book)); // sent by Bob
// adds new goal write(book)[source(bob)] for Tom
.send(tom,unachieve,write(book)); // sent by Bob
// removes goal write(book)[source(bob)] for Tom
```

### Communication of test goal

When an agent receives an **askOne** or **askAll** message, the content is a new test goal (annotated with the sender of the message)

```
.send(tom,askOne,published(P),Answer); // sent by Bob
// adds new goal ?publisher(P)[source(bob)] for Tom
// the response of Tom will unify with Answer
```

# Plans dynamics

## Agent Dynamics

The plans that form the plan library of the agent come from

- ▶ plans added (resp. removed) dynamically by intentions in internal reasoning:
  - ▶ `.add_plan` (resp. `.remove_plan`)
- ▶ plans added (resp. removed) by communication:
  - ▶ `tellHow` (resp. `untellHow`)

### Example

```
.send(bob, askHow, +!goto(_,_) [source(_)], ListOfPlans);  
...  
.plan_label(Plan, hp); // get a plans based on a plan's label  
.send(A, tellHow, Plan);  
.send(bob, tellHow, "+!start : true <- .println(hello)").
```

# A note about “Control”

Agents can control (manipulate) their own (and influence the others)

- ▶ beliefs
- ▶ goals
- ▶ plan

By doing so they control their behaviour

The developer provides initial values of these elements and thus also influence the behaviour of the agent

# Outline

Agent Abstractions

Agent Dynamics

Other language features

- Namespaces

- Strong Negation

- Prolog-like Rules

- Plan Annotations & Concurrent Plans

- Declarative Goal Patterns

- Meta Programming

Integrating **A** & **A** dimensions

Agent Management Infrastructure in JaCaMo

Comparison with other paradigms



# Outline

Agent Abstractions

Agent Dynamics

Other language features

- Namespaces

- Strong Negation

- Prolog-like Rules

- Plan Annotations & Concurrent Plans

- Declarative Goal Patterns

- Meta Programming

Integrating **A** & **A** dimensions

Agent Management Infrastructure in JaCaMo

Comparison with other paradigms

# Namespace

## Other language features

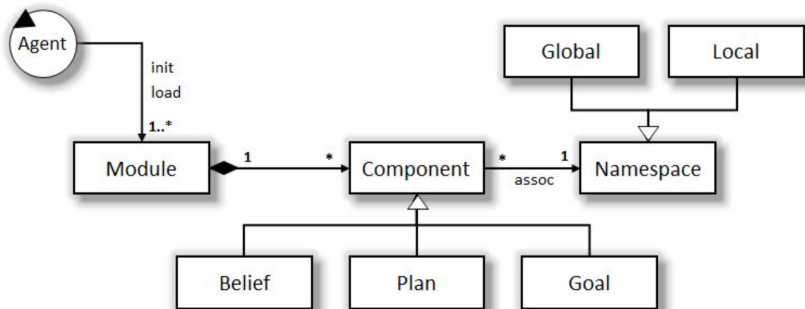
- ▶ Abstract container in the mind of agent, created to hold a logical grouping of beliefs, goals, events, plans and actions
- ▶ Identified by a name, used to prefix (using ::) the elements belonging to it:

`ns1::color(box,blue)` // color is in namespace ns1

- ▶ Two types:
  - ▶ Global namespace: any element associated with the global namespace can be consulted, changed by any other namespace
  - ▶ Local namespace: elements can only be used by the namespace
  - ▶ ↪ possibility of sharing elements by means of a common global namespace
- ▶ Namespace can be defined by:
  - ▶ module program of beliefs, goals and plans (i.e. a usual agent program).  
Every agent has one initial module (its initial program) into which other modules can be loaded
  - ▶ associating observable properties or actions of artifacts

# Modules and Namespaces

## Other language features



# Modules and Namespaces

## Other language features

## Inspection of agent **alice**

### - Beliefs

```
{include("initiator.asl", pc)}  
{include("initiator.asl", tv)}
```

```
!pc::startCNP(fix(pc)).  
!tv::startCNP(fix(tv)).
```

```
+pc::winner(X)  
  <- .print(X) .
```

```
tv::  
introduction(participant)[source(compan  
propose(11.075337225252543)[source  
propose(12.043311087442898)[source  
propose(12.81277904935436)[source  
winner(company_A1)[source(self)].
```

```
#8priv::  
state(finished)[source(self)].
```

```
pc::  
introduction(participant)[source(compan  
propose(11.389500048463455)[source  
propose(11.392553683771682)[source  
propose(12.348901000262853)[source  
winner(company_A2)[source(self)].
```

# Outline

Agent Abstractions

Agent Dynamics

Other language features

Namespaces

**Strong Negation**

Prolog-like Rules

Plan Annotations & Concurrent Plans

Declarative Goal Patterns

Meta Programming

Integrating **A** & **A** dimensions

Agent Management Infrastructure in JaCaMo

Comparison with other paradigms

# Strong Negation

## Other language features

```
+!leave(home)  
:   ~raining  
<- open(curains); ...
```

```
+!leave(home)  
:   not raining & not ~raining  
<- .send(mum,askOne,raining,Answer,3000); ...
```

# Outline

Agent Abstractions

Agent Dynamics

Other language features

Namespaces

Strong Negation

Prolog-like Rules

Plan Annotations & Concurrent Plans

Declarative Goal Patterns

Meta Programming

Integrating **A** & **A** dimensions

Agent Management Infrastructure in JaCaMo

Comparison with other paradigms

# Prolog-like Rules in the Belief Base

## Other language features

```
tall(X) :-  
    woman(X) & height(X, H) & H > 1.70  
    |  
    man(X) & height(X, H) & H > 1.80.  
  
likely_color(Obj,C) :-  
    colour(Obj,C)[degOfCert(D1)] &  
    not (colour(Obj,_)[degOfCert(D2)] & D2 > D1) &  
    not ~colour(C,B).
```



# Outline

Agent Abstractions

Agent Dynamics

Other language features

Namespaces

Strong Negation

Prolog-like Rules

Plan Annotations & Concurrent Plans

Declarative Goal Patterns

Meta Programming

Integrating **A** & **A** dimensions

Agent Management Infrastructure in JaCaMo

Comparison with other paradigms

# Plan Annotations

## Other language features

- ▶ Like beliefs, plans can also have **annotations**, which go in the **label**
- ▶ Annotations contain meta-level information for the plan, which selection functions can take into consideration
- ▶ The annotations in an intended plan instance can be changed **dynamically** (e.g. to change intention priorities)
- ▶ There are some pre-defined plan annotations, e.g. to force a breakpoint at that plan or to make the whole plan execute atomically

### Example (an annotated plan)

```
@myPlan[chance_of_success(0.3), usual_payoff(0.9),  
        any_other_property]  
+!g(X) : c(t) <- a(X).
```

# Concurrent Plans

## Other language features

► fork-join-and operator `|&|`

```
+!ga <- ...; !gb; ....  
+!gb <- ...; (!g1 |&| !g2); a1; ... // fork-join-and  
// a1 will be executed when !g2 and !g1 will be achieved  
► fork-join-xor operator |||
```

```
+!ga <- ...; !gb; ....  
+!gb <- ...; (!g1 ||| !g2); a1; ... // fork-join-xor  
// a1 will be executed after !g2 or !g1 are achieved  
// when one of !g2 or !g1 is achieved the other is dropped
```

```
-!g1 : true <- !g1. // in case of some failure  
-!g2 : true <- !g2. // in case of some failure  
+g1 : true <- .succeed_goal(g1).  
+g2 : true <- .succeed_goal(g2).  
+f1 : true <- .fail_goal(g1). // f1 drop condition for g1  
+f2 : true <- .fail_goal(g2). // f2 drop condition for g2
```

# Outline

Agent Abstractions

Agent Dynamics

Other language features

Namespaces

Strong Negation

Prolog-like Rules

Plan Annotations & Concurrent Plans

**Declarative Goal Patterns**

Meta Programming

Integrating **A** & **A** dimensions

Agent Management Infrastructure in JaCaMo

Comparison with other paradigms

# Declarative Goal Patterns: Achievement goal

Other language features

## Example (Example)

```
+!g : g <- true. // g declarative goal
```

```
+!g : c1 <- p1; ?g.
```

```
+!g : c2 <- p2; ?g.
```

```
...
```

```
+!g : cn <- pn; ?g.
```

```
+g : true <- .succeed__goal(g).
```

# Backtracking Declarative Goal Patterns

## Other language features

### Example (Example)

```
+!g : g <- true. // g declarative goal
```

```
+!g : c1 <- p1; ?g.
```

```
+!g : c2 <- p2; ?g.
```

```
...
```

```
+!g : cn <- pn; ?g.
```

```
+g : true <- .succeed__goal(g).
```

```
-!g : true <- !!g.
```

# Exclusive Backtracking Declarative Goal Pattern

Other language features

## Example (Example)

```
+!g : g <- true. // g declarative goal
```

```
+!g : not p(1,g) & c1 <- +p(1,g); p1; ?g.
```

```
+!g : not p(2,g) & c2 <- +p(2,g); p2; ?g.
```

```
...
```

```
+!g : not p(n,g) & cn <- +p(n,g); pn; ?g.
```

```
-?g : true <- !!g.
```

```
+g : true <- .abolish(p(_,g); .succeed_goal(g).
```

# Failure Handling: Contingency Plans

## Other language features

### Example (Example)

**!g1** // initial goal

**+!g1** : true <- **!g2(X)**; .print("end g1 ",X).

**+!g2** : true <- **!g3(X)**; .print("end g2 ",X).

**+!g3** : true <- **!g4(X)**; .print("end g3 ",X).

**+!g4** : true <- **!g5(X)**; .print("end g4 ",X).

**+!g5** : true <- .fail.

**-!g3(X)** : true <- .print("in g3 failure").



# Failure Handling: Contingency Plans

## Other language features

### Example (Example)

**!g1** // initial goal

**+!g1** : true <- **!g2(X)**; .print("end g1 ",X).

**+!g2** : true <- **!g3(X)**; .print("end g2 ",X).

**+!g3** : true <- **!g4(X)**; .print("end g3 ",X).

**+!g4** : true <- **!g5(X)**; .print("end g4 ",X).

**+!g5** : true <- .fail.

**-!g3(X)** : true <- .print("in g3 failure").

saying: in g3 failure

saying: end g2 failure

saying: end g1 failure

# Failure Handling: Contingency Plans

## Other language features

### Example (blind commitment to g)

```
+!g : g. // g is a declarative goal

+!g : ... <- a1; ?g.
+!g : ... <- a2; ?g.
+!g : ... <- a3; ?g.

+!g : true <- !g. // keep trying
-!g : true <- !g. // in case of some failure

+g : true <- .succeed_goal(g).
```

# Failure Handling: Contingency Plans

## Other language features

### Example (single minded commitment)

```
+!g : g. // g is a declarative goal

+!g : ... <- a1; ?g.
+!g : ... <- a2; ?g.
+!g : ... <- a3; ?g.

+!g : true <- !g. // keep trying
-!g : true <- !g. // in case of some failure

+g : true <- .succeed_goal(g).
+f : true <- .fail_goal(g). // f is the drop
condition for g
```

# Failure Handling: Compiler pre-processing – directives

## Other language features

### Example (single minded commitment)

```
{ begin smc(g,f)}  
  +!g : ... <- a1.  
  +!g : ... <- a2.  
  +!g : ... <- a3.  
{ end }
```

# Outline

Agent Abstractions

Agent Dynamics

Other language features

Namespaces

Strong Negation

Prolog-like Rules

Plan Annotations & Concurrent Plans

Declarative Goal Patterns

Meta Programming

Integrating **A** & **A** dimensions

Agent Management Infrastructure in JaCaMo

Comparison with other paradigms

# Meta Programming

## Other language features

### Example (an agent that asks for plans *on demand*)

```
-!G[error(no_relevant)] : teacher(T)
  <- .send(T, askHow, { +!G }, Plans);
    .add_plan(Plans);
    !G.
```

*in the event of a failure to achieve **any** goal **G** due to no relevant plan, asks a teacher for plans to achieve **G** and then try **G** again*

- ▶ The failure event is annotated with the error type, line, source, ...  
`error(no_relevant)` means no plan in the agent's plan library to achieve **G**
- ▶ `{ +!G }` is the syntax to enclose triggers/plans as terms

# Outline

Agent Abstractions

Agent Dynamics

Other language features

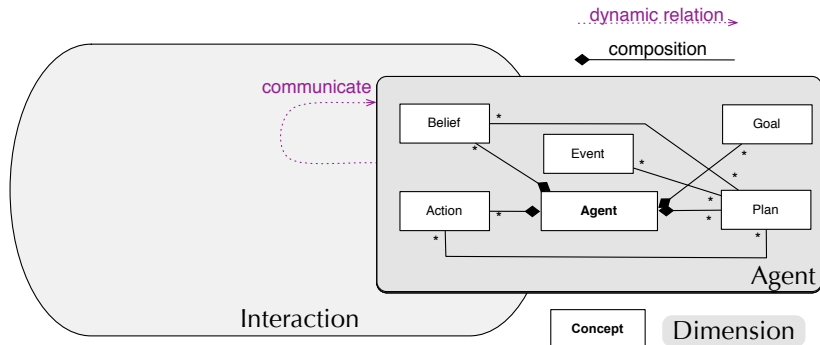
Integrating **A** & **A** dimensions

Agent Management Infrastructure in JaCaMo

Comparison with other paradigms

Conclusions and wrap-up

# Integrating A & A dimensions





# Communicative Actions

Use of the internal action `.send` with performative verbs and corresponding content:

- ▶ **tell**, **untell**: to share beliefs,
- ▶ **achieve**, **unachieve**: to delegate achievement goal,
- ▶ **askOne**, **askAll**: to delegate test goal,
- ▶ **askHow**: to request plans,
- ▶ **tellHow**, **untellHow**: to share plans.

# Outline

Agent Abstractions

Agent Dynamics

Other language features

Integrating **A** & **A** dimensions

Agent Management Infrastructure in JaCaMo

Comparison with other paradigms

Conclusions and wrap-up

# Jason Customisations

- ▶ **Agent** class customisation:  
selectMessage, selectEvent, selectOption, selectIntention, buf, brf, ...
- ▶ Agent **architecture** customisation:  
perceive, act, sendMsg, checkMail, ...
- ▶ **Belief base** customisation:  
add, remove, contains, ...
  - ▶ Example available with *Jason*: persistent belief base (in text files, in data bases, ...)

# Outline

Agent Abstractions

Agent Dynamics

Other language features

Integrating **A** & **A** dimensions

Agent Management Infrastructure in JaCaMo

Comparison with other paradigms

Conclusions and wrap-up

Consider a very simple robot with two goals:

- ▶ when a piece of gold is seen, go to it
- ▶ when battery is low, go charge it

## Java code – go to gold

```
public class Robot extends Thread {  
    boolean seeGold, lowBattery;  
    public void run() {  
        while (true) {  
            while (! seeGold) {  
                a = randomDirection();  
                doAction(go(a));  
            }  
            while (seeGold) {  
                a = selectDirection();  
                doAction(go(a));  
            }  
        }  
    }  
}
```

## Java code – charge battery

```
public class Robot extends Thread {
    boolean seeGold, lowBattery;
    public void run() {
        while (true) {
            while (! seeGold) {
                a = randomDirection();
                doAction(go(a));
                if (lowBattery) charge();
            }
            while (seeGold) {
                a = selectDirection ();
                if (lowBattery) charge();
                doAction(go(a));
                if (lowBattery) charge();
            }
        }
    }
}
```

## Jason code

```
direction(gold)    :- see(gold).
direction(random)  :- not see(gold).

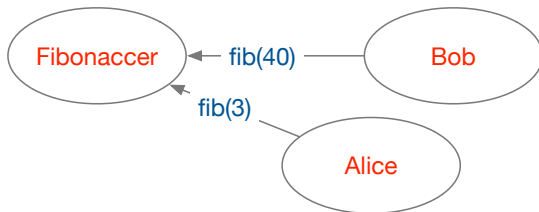
+!find(gold)                // long term goal
    <- ?direction(A);
        go(A);
        !find(gold).

+battery(low)              // reactivity
    <- !charge.

^!charge[state(started)]    // goal meta-events
    <- .suspend(find(gold)).
^!charge[state(finished)]
    <- .resume(find(gold)).
```



# Fibonacci calculator server – “java” version

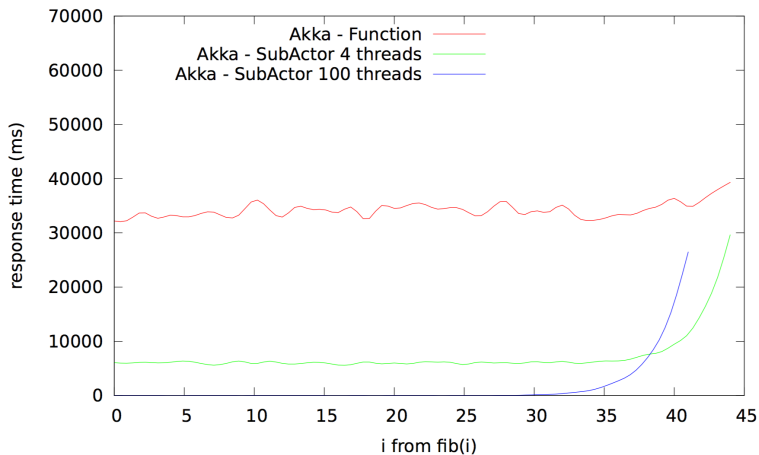


```
while true
  m = receiveMsg()
  if m == fib(N)
    m.answer(fib(m.getArg(0)))
  ...
```

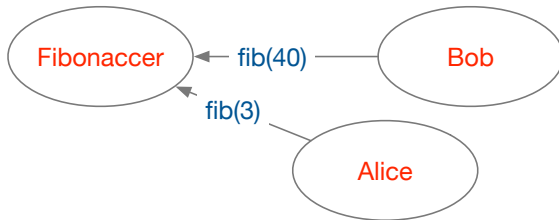
```
int fib(int n)
  if n <= 2
    return 1
  else
    return fib(n-1)+fib(n-2)
```

How long will Alice wait?

# Fibonacci calculator server – Akka



## Fibonacci calculator agent – Jason version



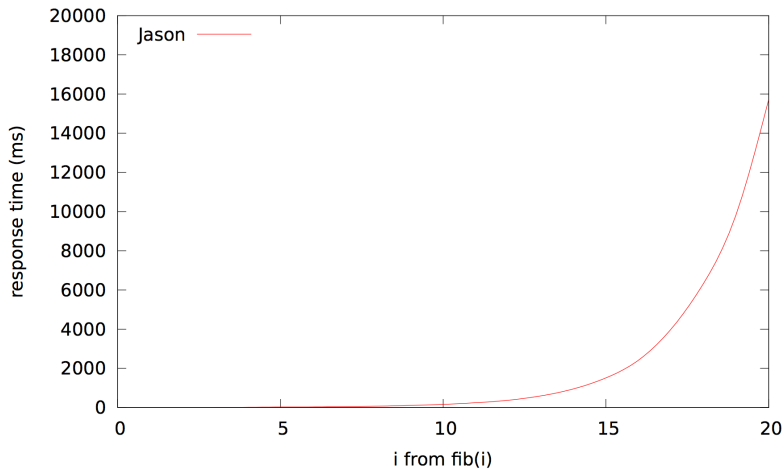
`+?fib(1,1).`

`+?fib(2,1).`

`+?fib(N,F) <- ?fib(N-1,A); ?fib(N-2,B); F = A+B.`

How long will Alice wait?

## Fibonacci calculator agent – Jason version



# Jason × Prolog

- ▶ With the *Jason* extensions, nice separation of theoretical and **practical reasoning**
- ▶ BDI architecture allows
  - ▶ long-term goals (goal-based behaviour)
  - ▶ reacting to changes in a dynamic environment
  - ▶ handling multiple foci of attention (concurrency)
- ▶ Acting on an environment and a higher-level conception of a distributed system

# Outline

Agent Abstractions

Agent Dynamics

Other language features

Integrating **A** & **A** dimensions

Agent Management Infrastructure in JaCaMo

Comparison with other paradigms

Conclusions and wrap-up

# Some Shortfalls

- ▶ **IDEs** and programming tools are still not anywhere near the level of OO languages
- ▶ **Debugging** is a serious issue — much more than “mind tracing” is needed
- ▶ Combination with **organisational** models is very recent — much work still needed
- ▶ Principles for using **declarative goals** in practical programming problems still not “textbook”
- ▶ Large applications and **real-world** experience much needed!

# Some Trends

- ▶ **Modularity** and encapsulation
  - ▶ **Debugging** MAS is hard: problems of concurrency, simulated environments, emergent behaviour, mental attitudes
  - ▶ Logics for Agent Programming languages
  - ▶ Further work on combining with interaction, environments, and organisations
  - ▶ We need to put everything together: rational agents, environments, organisations, normative systems, reputation systems, economically inspired techniques, etc.
- ~▶ **Multi-Agent Programming**



# Some Related Projects I

- ▶ **Speech-act** based communication  
Joint work with Renata Vieira, Álvaro Moreira, and Mike Wooldridge
- ▶ **Cooperative** plan exchange  
Joint work with Viviana Mascardi, Davide Ancona
- ▶ **Plan Patterns** for Declarative Goals  
Joint work with M.Wooldridge
- ▶ **Planning** (Felipe Meneguzzi and Colleagues)
- ▶ **Web and Mobile Applications** (Alessandro Ricci and Colleagues)
- ▶ **Belief Revision**  
Joint work with Natasha Alechina, Brian Logan, Mark Jago

# Some Related Projects II

- ▶ **Ontological** Reasoning
  - ▶ Joint work with Renata Vieira, Álvaro Moreira
  - ▶ **JASDL**: joint work with Tom Klapiscak
- ▶ Goal-Plan Tree Problem (Thangarajah et al.)  
Joint work with Tricia Shaw
- ▶ Trust reasoning (ForTrust project)
- ▶ Agent verification and model checking  
Joint project with M.Fisher, M.Wooldridge, W.Visser, L.Dennis, B.Farwer

# Some Related Projects III

- ▶ Environments, Organisation and Norms
  - ▶ Normative environments  
Join work with A.C.Rocha Costa and F.Okuyama
  - ▶ MADeM integration (Francisco Grimaldo Moreno)
  - ▶ Normative integration (Felipe Meneguzzi)
- ▶ More on `jason.sourceforge.net`, related projects

# Summary

- ▶ **AgentSpeak**

- ▶ Logic + BDI
- ▶ Agent programming language

- ▶ **Jason**

- ▶ AgentSpeak interpreter
- ▶ Implements the operational semantics of AgentSpeak
- ▶ Speech-act based communication
- ▶ Highly customisable
- ▶ Useful tools
- ▶ Open source
- ▶ Open issues

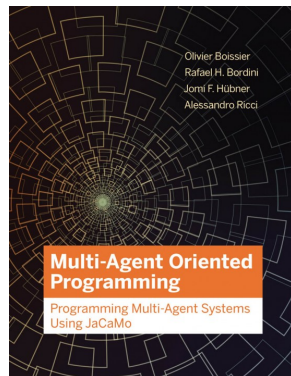
# Further Resources

- ▶ <http://jason.sourceforge.net>
- ▶ R.H. Bordini, J.F. Hübner, and M. Wooldrige  
**Programming Multi-Agent Systems in AgentSpeak using Jason**  
John Wiley & Sons, 2007.



# Further Resources

- ▶ <http://jacamo.sourceforge.net>
- ▶ O. Boissier, R.H. Bordini, J.F. Hübner, and A. Ricci  
**Multi-Agent Oriented Programming:  
Programming Multi-Agent Systems  
Using JaCaMo**  
MIT Press, 2020.



# Bibliography I



Boissier, O., Bordini, R., Hübner, J. F., and Ricci, A. (2020).  
*Multi-Agent Oriented Programming: Programming Multi-Agent Systems Using JaCaMo*.  
The MIT Press.



Boissier, O., Bordini, R. H., Hübner, J. F., Ricci, A., and Santi, A. (2011).  
Multi-agent oriented programming with jacamo.  
*Science of Computer Programming*, pages –.



Bordini, R. H., Hübner, J. F., and Wooldrige, M. (2007).  
*Programming Multi-Agent Systems in AgentSpeak using Jason*.  
Wiley Series in Agent Technology. John Wiley & Sons.



d'Inverno, M., Kinny, D., Luck, M., and Wooldridge, M. (1997).  
A formal specification of dmars.  
In *International Workshop on Agent Theories, Architectures, and Languages*,  
pages 155–176. Springer.



Georgeff, M. P. and Lansky, A. L. (1987).  
Reactive reasoning and planning.  
In *AAAI*, volume 87, pages 677–682.

# Bibliography II



Hübner, J. F., Boissier, O., Kitio, R., and Ricci, A. (2009).

Instrumenting Multi-Agent Organisations with Organisational Artifacts and Agents.

*Journal of Autonomous Agents and Multi-Agent Systems.*



Rao, A. S. (1996).

Agentspeak(I): Bdi agents speak out in a logical computable language.

In de Velde, W. V. and Perram, J. W., editors, *MAAMAW*, volume 1038 of *Lecture Notes in Computer Science*, pages 42–55. Springer.



Rao, A. S., Georgeff, M. P., et al. (1995).

Bdi agents: From theory to practice.

In *ICMAS*, volume 95, pages 312–319.



Ricci, A., Piunti, M., Viroli, M., and Omicini, A. (2009).

Environment programming in CArtAgO.

In *Multi-Agent Programming: Languages, Platforms and Applications, Vol.2*. Springer.