Agent Oriented Programming with Jason

Jomi F. Hübner

Federal University of Santa Catarina, Brazil

PPGEAS 2017 — UFSC

Outline

- Introduction
- BDI architecture
- *Jason* hello world
- *Jason* (details)
- Conclusions

(slides written together with R. Bordini, O. Boissier, and A. Ricci)

Multi-Agent System (our perspective)

def...

An organisation of autonomous agents interacting together within a shared environment

- agents can be: software/hardware, coarse-grain/small-grain, heterogeneous/homogeneous, reactive/pro-active entities
- environment can be virtual/physical, passive/active, deterministic/non deterministic, ...
- interaction is the motor of dynamic in MAS. Interaction can be: direct/indirect between agents, interaction between agent and environment
- organisation can be pre-defined/emergent, static/adaptive, open/closed, ...

Multi-Agent System (our perspective)

def...

An organisation of autonomous agents interacting together within a shared environment

- agents can be: software/hardware, coarse-grain/small-grain, heterogeneous/homogeneous, reactive/pro-active entities
- environment can be virtual/physical, passive/active, deterministic/non deterministic, ...
- interaction is the motor of dynamic in MAS. Interaction can be: direct/indirect between agents, interaction between agent and environment
- organisation can be pre-defined/emergent, static/adaptive, open/closed, ...

Multi-Agent System (our perspective)

def...

An organisation of autonomous agents interacting together within a shared environment

- agents can be: software/hardware, coarse-grain/small-grain, heterogeneous/homogeneous, reactive/pro-active entities
- environment can be virtual/physical, passive/active, deterministic/non deterministic, ...
- interaction is the motor of dynamic in MAS. Interaction can be: direct/indirect between agents, interaction between agent and environment
- organisation can be pre-defined/emergent, static/adaptive, open/closed, ...

Multi-Agent System (our perspective)

def...

An organisation of autonomous agents interacting together within a shared environment

- agents can be: software/hardware, coarse-grain/small-grain, heterogeneous/homogeneous, reactive/pro-active entities
- environment can be virtual/physical, passive/active, deterministic/non deterministic, ...
- interaction is the motor of dynamic in MAS. Interaction can be: direct/indirect between agents, interaction between agent and environment
- organisation can be pre-defined/emergent, static/adaptive, open/closed, ...

Multi-Agent System (our perspective)

def...

An organisation of autonomous agents interacting together within a shared environment

- agents can be: software/hardware, coarse-grain/small-grain, heterogeneous/homogeneous, reactive/pro-active entities
- environment can be virtual/physical, passive/active, deterministic/non deterministic, ...
- interaction is the motor of dynamic in MAS. Interaction can be: direct/indirect between agents, interaction between agent and environment
- organisation can be pre-defined/emergent, static/adaptive, open/closed, ...

Multi-Agent System (our perspective)

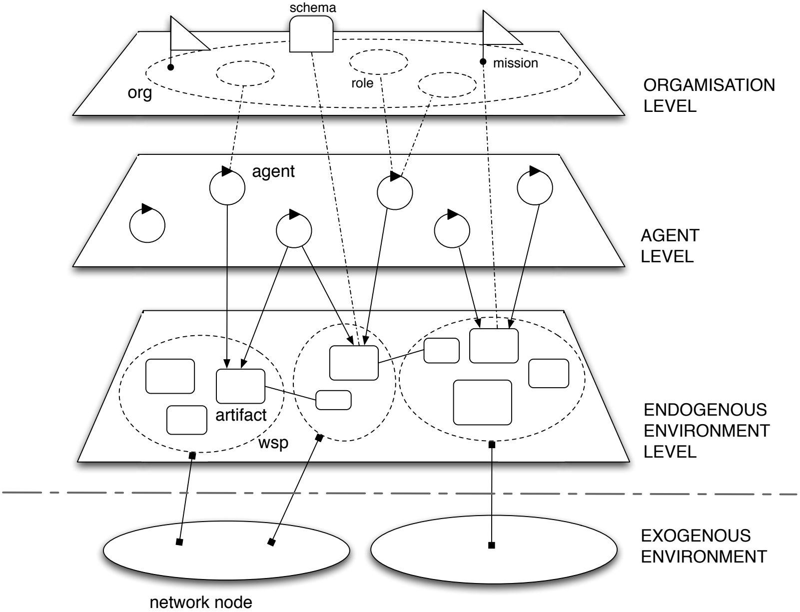
def...

An organisation of autonomous agents interacting together within a shared environment

MAS is not a simple set of agents

- agents can be: software/hardware, coarse-grain/small-grain, heterogeneous/homogeneous, reactive/pro-active entities
- environment can be virtual/physical, passive/active, deterministic/non deterministic, ...
- interaction is the motor of dynamic in MAS. Interaction can be: direct/indirect between agents, interaction between agent and environment
- organisation can be pre-defined/emergent, static/adaptive, open/closed, ...

Levels in Multi-Agent Systems



Abstractions in Multi-Agent Systems

- Individual level
  - autonomy, situatedness
  - beliefs, desires, goals, intentions, plans
  - sense/reason/act, reactive/pro-active behaviour
- Environment level
  - resources and services that agents can access and control
  - sense/act
- Social level
  - cooperation, languages, protocols
- Organisation level
  - coordination, regulation patterns, norms, obligations, rights

Agent Oriented Programming

— **AOP** —

Literature

Books: [Bordini et al., 2005], [Bordini et al., 2009]

Proceedings: ProMAS, DALT, LADS, EMAS, AGERE, ...

Surveys: [Bordini et al., 2006], [Fisher et al., 2007] ...

Languages of historical importance: Agent0 [Shoham, 1993],
AgentSpeak(L) [Rao, 1996],
MetateM [Fisher, 2005],
3APL [Hindriks et al., 1997],
Golog [Giacomo et al., 2000]

Other prominent languages:
Jason [Bordini et al., 2007],
Jadex [Pokahr et al., 2005], 2APL [Dastani, 2008],
GOAL [Hindriks, 2009], JACK [Winikoff, 2005],
JIAC, ASTRA

But many others languages and platforms...

Some Languages and Platforms

Jason (Hübner, Bordini, ...); 3APL and 2APL (Dastani, van Riemsdijk, Meyer, Hindriks, ...); Jadex (Braubach, Pokahr); MetateM (Fisher, Guidini, Hirsch, ...); ConGoLog (Lesperance, Levesque, ... / Boutilier – DTGolog); Teamcore/ MTDP (Milind Tambe, ...); IMPACT (Subrahmanian, Kraus, Dix, Eiter); CLAIM (Amal El Fallah-Seghrouchni, ...); GOAL (Hindriks); BRAHMS (Sierhuis, ...); SemantiCore (Blois, ...); STAPLE (Kumar, Cohen, Huber); Go! (Clark, McCabe); Bach (John Lloyd, ...); MINERVA (Leite, ...); SOCS (Torroni, Stathis, Toni, ...); FLUX (Thielscher); JIAC (Hirsch, ...); JADE (Agostino Poggi, ...); JACK (AOS); Agentis (Agentis Software); Jackdaw (Calico Jack); ASTRA (Rem Collier); SARL (Stephane Galland); *simpAL*, *ALOO* (Ricci, ...);

...

Agent Oriented Programming

Features

- Reacting to events $\times$ long-term goals
- Course of actions depends on circumstance
- Plan failure (dynamic environments)
- Social ability
- Combination of theoretical and practical reasoning

Agent Oriented Programming

Fundamentals

- Use of mentalistic notions and a societal view of computation [Shoham, 1993]
- Heavily influence by the BDI architecture and reactive planning systems [Bratman et al., 1988]

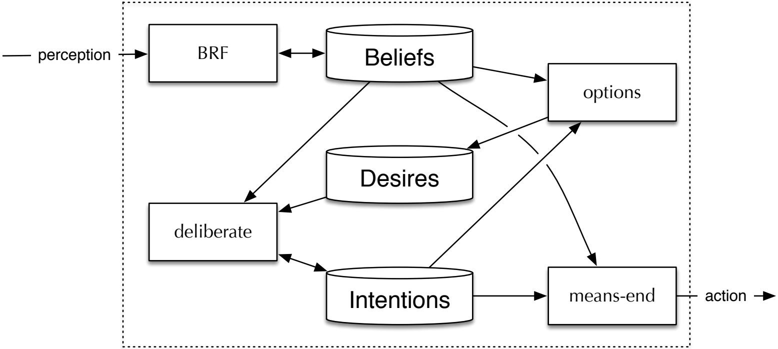
Motivation for BDI — autonomous robot

[Cohen and Levesque, 1990]

household robot.[1] You say "Willie, bring me a beer." The robot replies "OK, boss." Twenty minutes later, you screech "Willie, why didn't you bring that beer?" It answers "Well, I intended to get you the beer, but I decided to do something else." Miffed, you send the wise guy back to the manufacturer, complaining about a lack of commitment. After retrofitting, Willie is returned, marked "Model C: The Committed Assistant." Again, you ask Willie to bring a beer. Again, it accedes, replying "Sure thing." Then you ask: "What kind did you buy?" It answers: "Genessee." You say "Never mind." One minute later, Willie trundles over with a Genessee in its gripper. This time, you angrily return Willie for overcommitment. After still more tinkering, the manufacturer sends Willie back, promising no more problems with its commitments. So, being a somewhat trusting consumer, you accept the rascal back into your household, but as a test, you ask it to bring you your last beer. Willie again accedes, saying "Yes, Sir." (Its attitude problem seems to have been fixed.) The robot gets the beer and starts towards you. As it approaches, it lifts its arm, wheels around, deliberately smashes the bottle, and trundles off. Back at the plant, when interrogated by customer service as to why it had abandoned its commitments, the robot replies that according to its specifications, it kept its commitments as long as required—commitments must be dropped when fulfilled or impossible to achieve. By smashing the last bottle, the commitment

BDI architecture

(the mentalistic view)



BDI architecture [Wooldridge, 2009]

```
while true do
    B ← brf(B, perception()) ;          // belief revision
    D ← options(B, I) ;                 // desire revision
    I ← deliberate(B, D, I) ;           // get intentions
    π ← meansend(B, I, A) ;             // gets a plan
    while π ≠ ∅ do
        execute( head(π) )
        π ← tail(π)
```

BDI architecture [Wooldridge, 2009]

while true **do**
    $B \leftarrow brf(B, perception())$; `// belief revision`
    $D \leftarrow options(B, I)$; `// desire revision`
    $I \leftarrow deliberate(B, D, I)$; `// get intentions`
    $\pi \leftarrow meansend(B, I, A)$; `// gets a plan`
    while $\pi \neq \emptyset$ **do**
        execute($head(\pi)$)
        $\pi \leftarrow tail(\pi)$

fine for pro-activity, but not for reactivity (over commitment)

BDI architecture [Wooldridge, 2009]

```
while true do
    B ← brf(B, perception()) ;                    // belief revision
    D ← options(B, I) ;                           // desire revision
    I ← deliberate(B, D, I) ;                     // get intentions
    π ← meansend(B, I, A) ;                       // gets a plan
    while π ≠ ∅ do
        execute( head(π) )
        π ← tail(π)
        B ← brf(B, perception())
        if ¬sound(π, I, B) then
            π ← meansend(B, I, A)
```

revise commitment to plan – re-planning for context adaptation

BDI architecture [Wooldridge, 2009]

```
while true do
    B ← brf(B, perception()) ;                              // belief revision
    D ← options(B, I) ;                                     // desire revision
    I ← deliberate(B, D, I) ;                               // get intentions
    π ← meansend(B, I, A) ;                                 // gets a plan
    while π ≠ ∅ and ¬succeeded(I, B) and ¬impossible(I, B) do
        execute( head(π) )
        π ← tail(π)
        B ← brf(B, perception())
        if ¬sound(π, I, B) then
            π ← meansend(B, I, A)
```

revise commitment to intentions – Single-Minded Commitment

BDI architecture [Wooldridge, 2009]

```
while true do
    B ← brf(B, perception()) ;                              // belief revision
    D ← options(B, I) ;                                     // desire revision
    I ← deliberate(B, D, I) ;                               // get intentions
    π ← meansend(B, I, A) ;                                 // gets a plan
    while π ≠ ∅ and ¬succeeded(I, B) and ¬impossible(I, B) do
        execute( head(π) )
        π ← tail(π)
        B ← brf(B, perception())
        if reconsider(I, B) then
            D ← options(B, I)
            I ← deliberation(B, D, I)
        if ¬sound(π, I, B) then
            π ← meansend(B, I, A)
```

reconsider the intentions (not always!)

Intentions

- Intentions pose problems for the agents: they need to determine a way to achieve them
(planning and acting)
- Intentions provide a "screen of admissibility" for adopting new intentions
- Agents keep tracking their success of attempting to achieve their intentions
- Agents should not spend all their time revising intentions (losing pro-activity and reactivity)

Jason

(let's go programming those nice concepts)

(BDI & Jason) Hello World – agent bob

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

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

(BDI & Jason) Hello World – agent bob

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

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

beliefs

- prolog like (FOL)

(BDI & Jason) Hello World – agent bob

```
happy(bob).
!say(hello).

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

desires

- prolog like
- with ! prefix

(BDI & Jason) Hello World – agent bob

```
happy(bob).
!say(hello).

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

plans

- define when a desire becomes an intention
  ⇝ deliberate
- how it is satisfied
- are used for practical reasoning
  ⇝ means-end

Hello World

desires from perception — options

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

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

Hello World

source of beliefs

```
+happy(bob)[source(A)]
   :  someone_who_knows_me_very_well(A)
   <- !say(hello).

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

Hello World

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).
```

Hello World

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).
```

Hello World

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)).
```

Hello World

intention revision

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

features

- we can have several intentions based on the same plans
  ⇝ running concurrently
- long term goals running
  ⇝ reaction meanwhile

AgentSpeak

The foundational language for *Jason*

- Originally proposed by Rao [Rao, 1996]
- Programming language for BDI agents
- Elegant notation, based on logic programming
- Inspired by PRS (Georgeff & Lansky), dMARS (Kinny), and BDI Logics (Rao & Georgeff)
- 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

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

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

Intentions: plans instantiated to achieve some goal

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

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

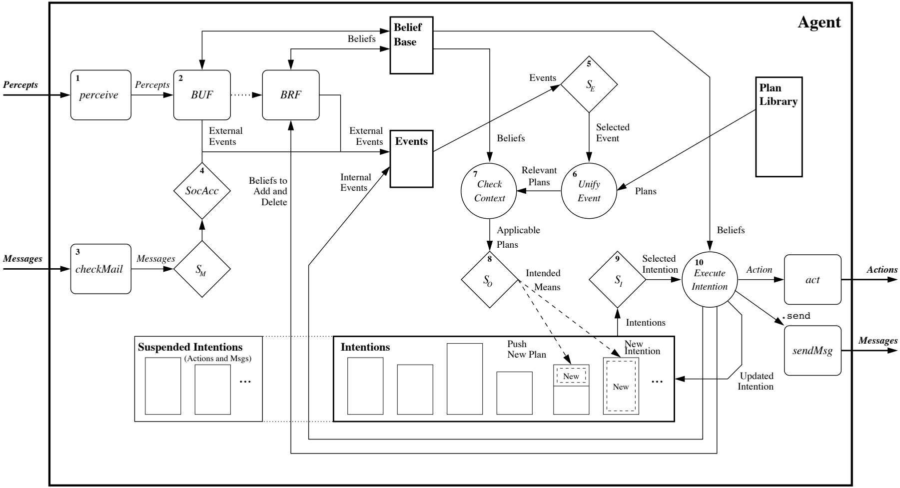
Intentions: plans instantiated to achieve some goal

Basic Reasoning cycle

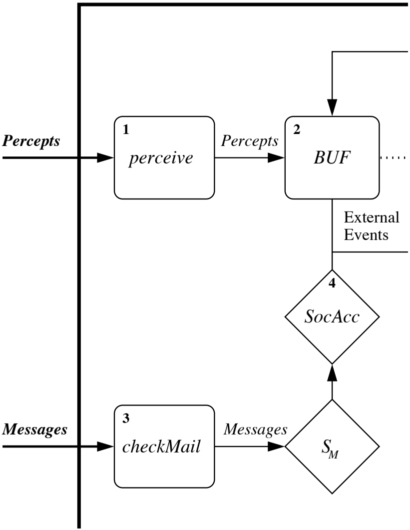
runtime interpreter

- 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

Jason Reasoning Cycle

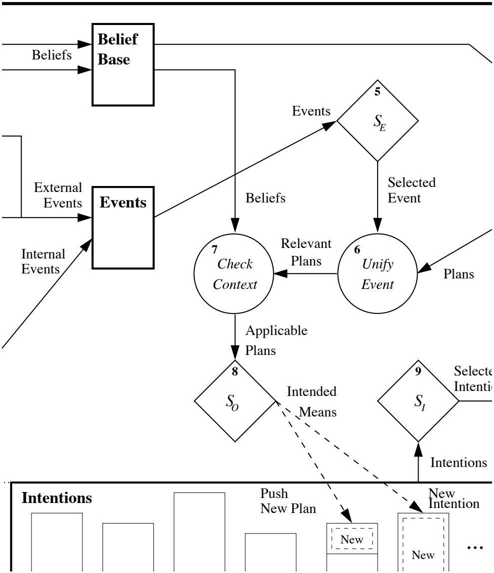


Jason Reasoning Cycle



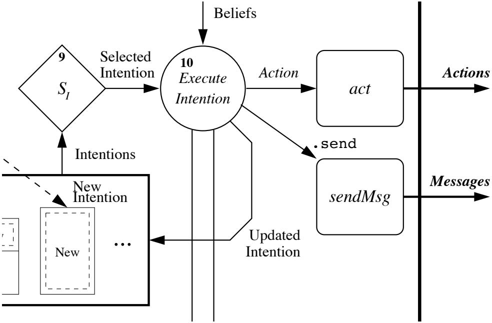
- machine perception
- belief revison
- knowledge representation
- communication, argumentation
- trust
- social power

Jason Reasoning Cycle



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

Jason Reasoning Cycle



- intention reconsideration
- scheduling
- action theories

Beliefs — Representation

Syntax

Beliefs are represented by annotated literals of first order logic

`functor(`$term_1$`, ..., `$term_n$`)[`$annot_1$`, ..., `$annot_m$`]`

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)].
```

Beliefs — Dynamics I

by perception

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

by intention

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)]
```

Beliefs — Dynamics II

by communication

when an agent receives a tell message, the content is a new belief annotated with the sender of the message

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

Goals — Representation

Types of goals

- Achievement goal: goal to do
- Test goal: goal to know

Syntax

Goals have the same syntax as beliefs, but are prefixed by
! (achievement goal) or
? (test goal)

Example (Initial goal of agent Tom)

```
!write(book).
```

Goals — Dynamics I

by intention

the plan operators **!** and **?** can be 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 II

by communication – 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
```

Goals — Dynamics III

by communication – 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
```

Triggering Events — Representation

- Events happen as consequence to changes in the agent's beliefs or goals
- An agent reacts to events by executing plans
- Types of plan triggering events

+b	(belief addition)
-b	(belief deletion)
+!g	(achievement-goal addition)
-!g	(achievement-goal deletion)
+?g	(test-goal addition)
-?g	(test-goal deletion)

Plans — Representation

An AgentSpeak plan has the following general structure:

```
triggering_event : context <- body.
```

where:

- the triggering event denotes the events that the plan is meant to handle
- the context represent the circumstances in which the plan can be used
- the body is the course of action to be used to handle the event if the context is believed true at the time a plan is being chosen to handle the event

Plans — Operators for Plan **Context**

Boolean operators

&	(and)
|	(or)
not	(not)
=	(unification)
>, >=	(relational)
<, <=	(relational)
==	(equals)
\\==	(different)

Arithmetic operators

+	(sum)
-	(subtraction)
*	(multiply)
/	(divide)
div	(divide – integer)
mod	(remainder)
**	(power)

Plans — Operators for Plan **Body**

```
+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 — 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).
```

Plans — Dynamics

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

- initial plans defined by the programmer
- plans added dynamically and intentionally by
  - .add_plan
  - .remove_plan
- plans received from
  - tellHow messages
  - untellHow

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

Failure Handling: Contingency Plans

Example (an agent blindly committed to g)

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

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

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

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

Failure Handling: Contingency Plans

Example (single minded commitment)

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

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

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

+g <-.succeed_goal(g).

+f : .super_goal(g,SG) <-.fail_goal(SG).
```

f is the drop condition for goal *g*

Compiler pre-processing – directives

Example (single minded commitment)

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

Meta Programming

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

Other Language Features

Strong Negation

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

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

Prolog-like Rules in the Belief Base

```
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(Obj,C).
```

Plan Annotations

- Like beliefs, plans can also have annotations, which go in the plan 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).
```

Internal Actions

- Unlike actions, internal actions do not change the environment
- Code to be executed as part of the agent reasoning cycle
- AgentSpeak is meant as a high-level language for the agent's practical reasoning and internal actions can be used for invoking legacy code elegantly

- Internal actions can be defined by the user in Java

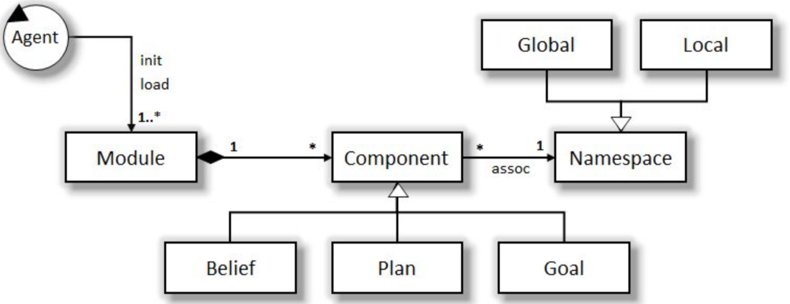
```
libname.action_name(...)
```

Standard Internal Actions

- Standard (pre-defined) internal actions have an empty library name
  - `.print(`*term*$_1$, *term*$_2$, ...`)`
  - `.union(`*list*$_1$, *list*$_2$, *list*$_3$`)`
  - `.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.

Namespaces & Modularity



Namespaces & Modularity

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

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

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

Inspection of agent **alice**

- Beliefs

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

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

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

Concurrent Plans

```
+!ga <- ...; !gb; ...
+!gb <- ...; !g1 |&| !g2; a1; ...

+!ga <- ...; !gb; ...
+!gb <- ...; !g1 ||| !g2; a1; ...

+!g <- x; (a;b) |&| (c;d) ||| (e;f); y.
```

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, ...)

Jason × Java

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();
} } }  }
```

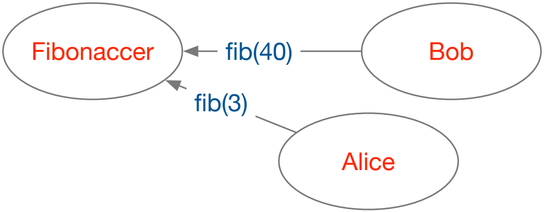
Jason code

```
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

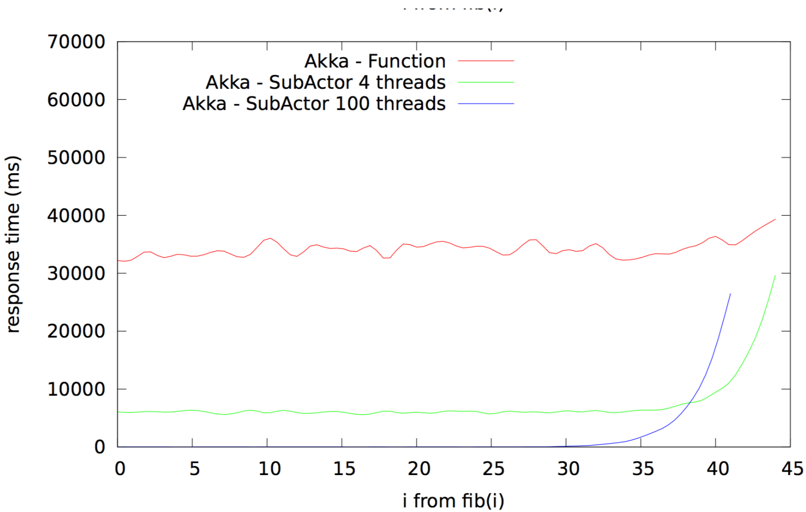



```
while true
  m = receiveMsg()
  if m == fib(N)
     m.answer(fib(m.getArg(0)))
  ...
```

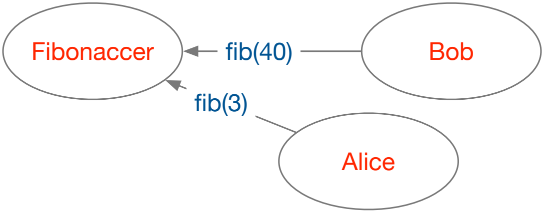
```
int fib(int n)
   if n <= 2
     return 1
   else
     return fib(n-1)+fib(n-2)
```

How long will Alice wait?

Fibonacci calculator server – Akka



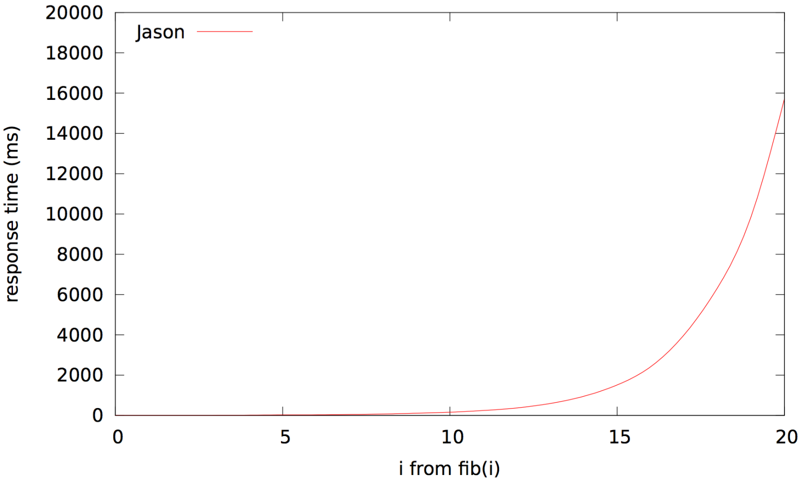
Fibonacci calculator agent – version Jason




```
+?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 server – Jason



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

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.

$\rightsquigarrow$ 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

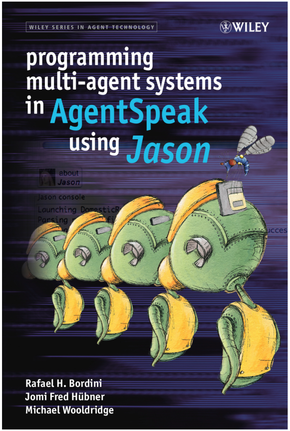
- AgentSpeak
  - Logic + BDI
  - Agent programming language
- *Jason*
  - AgentSpeak interpreter
  - Implements the operational semantics of AgentSpeak
  - Speech-act based communicaiton
  - Highly customisable
  - Useful tools
  - Open source
  - Open issues

Acknowledgements

- Many thanks to the
  - Various colleagues acknowledged/referenced throughout these slides
  - *Jason* users for helpful feedback
  - CNPq for supporting some of our current researh

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.



Bibliography I

Bordini, R. H., Braubach, L., Dastani, M., Fallah-Seghrouchni, A. E., Gómez-Sanz, J. J., Leite, J., O'Hare, G. M. P., Pokahr, A., and Ricci, A. (2006).
A survey of programming languages and platforms for multi-agent systems.
Informatica (Slovenia), 30(1):33–44.

Bordini, R. H., Dastani, M., Dix, J., and Fallah-Seghrouchni, A. E., editors (2005).
Multi-Agent Programming: Languages, Platforms and Applications, volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*.
Springer.

Bordini, R. H., Dastani, M., Dix, J., and Fallah-Seghrouchni, A. E., editors (2009).
Multi-Agent Programming: Languages, Tools and Applications.
Springer.

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

Bibliography II

Bratman, M. E., Israel, D. J., and Pollack, M. E. (1988).
Plans and resource-bounded practical reasoning.
Computational Intelligence, 4:349–355.

Cohen, P. R. and Levesque, H. J. (1990).
Intention is choice with commitment.
Artificial Intelligence, 42:213–261.

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

Fisher, M. (2005).
Metatem: The story so far.
In *PROMAS*, pages 3–22.

Fisher, M., Bordini, R. H., Hirsch, B., and Torroni, P. (2007).
Computational logics and agents: A road map of current technologies and future trends.
Computational Intelligence, 23(1):61–91.

Bibliography III

Giacomo, G. D., Lespérance, Y., and Levesque, H. J. (2000).
Congolog, a concurrent programming language based on the situation calculus.
Artif. Intell., 121(1-2):109–169.

Hindriks, K. V. (2009).
Programming rational agents in GOAL.
In [Bordini et al., 2009], pages 119–157.

Hindriks, K. V., de Boer, F. S., van der Hoek, W., and Meyer, J.-J. C. (1997).
Formal semantics for an abstract agent programming language.
In Singh, M. P., Rao, A. S., and Wooldridge, M., editors, *ATAL*, volume 1365 of *Lecture Notes in Computer Science*, pages 215–229. Springer.

Pokahr, A., Braubach, L., and Lamersdorf, W. (2005).
Jadex: A bdi reasoning engine.
In [Bordini et al., 2005], pages 149–174.

Rao, A. S. (1996).
Agentspeak(l): 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.

Bibliography IV

Shoham, Y. (1993).
Agent-oriented programming.
Artif. Intell., 60(1):51–92.

Winikoff, M. (2005).
Jack intelligent agents: An industrial strength platform.
In [Bordini et al., 2005], pages 175–193.

Wooldridge, M. (2009).
An Introduction to MultiAgent Systems.
John Wiley and Sons, 2nd edition.

TOC