

Programming agents with **Jason**

Jomi F. Hübner

ENS Mines Saint Etienne, France
`hubner@emse.fr`

(based on slides written by Bordini, Piunti, ...)

Outline

- 1 Origins and Fundamentals
- 2 Features
- 3 Multi-Agent System definition
- 4 Current Shortfalls and Future Trends

Agent Oriented Programming

Features

- *Reacting* to events $\times$ *long-term* goals
- Commit to courses of action *as late as possible* and dependent of the circumstances
- *Plan failure* (dynamic environments)
- *Rational* behaviour
- *Social* ability
- Combination of *theoretical* and *practical* reasoning
- E.g. of the best known and publicly available languages
 - Jadex (Pokahr, Braubach) <http://jadex.sf.net>
 - 2APL (Dastani, Meyer) <http://www.cs.uu.nl/2apl>
 - Jason (Bordini, Hübner) <http://jason.sf.net>

Agent Oriented Programming

Features

- *Reacting* to events $\times$ *long-term* goals
- Commit to courses of action *as late as possible* and dependent of the circumstances
- *Plan failure* (dynamic environments)
- *Rational* behaviour
- *Social* ability
- Combination of *theoretical* and *practical* reasoning
- E.g. of the best known and publicly available languages
 - Jadex (Pokahr, Braubach) <http://jadex.sf.net>
 - 2APL (Dastani, Meyer) <http://www.cs.uu.nl/2apl>
 - Jason (Bordini, Hübner) <http://jason.sf.net>

AgentSpeak

the foundational language for *Jason*

- Originally proposed by 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 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*
- Jointly developed with *Rafael H. Bordini*

Main concepts

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

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

Plans: are recipes for action, representing the agent's know-how

Intentions: plans instantiated to achieve some goal

Beliefs representation

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)] .  
liar(alice) [source(self),source(bob)] .  
~liar(bob) [source(self)] .
```

Changes in the belief base I

by perception

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

by intention

the operators `+` and `-` can be used to add and remove beliefs annotated with `source(self)`

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

Changes in the belief base 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

Types

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

Syntax

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

Example (initial goal of agent Tom)

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

New goals I

by intention

the 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);  
...
```

New goals II

by communication – achieve 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
```

New goals 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
```

Events

- Events happen as a consequence to changes in the agent's beliefs or goals
- Types of events
 - +b (belief addition)
 - b (belief deletion)
 - +!g (achievement-goal addition)
 - !g (achievement-goal deletion)
 - +?g (test-goal addition)
 - ?g (test-goal deletion)
- An agent reacts to events by executing plans

Plan library

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

- initial plans defined by the programmer
- plans added dynamically and intentionally by `.add_plan` (resp. `.remove_plan`)
- plans received from *tellHow* messages (resp. *untellHow*)

Plans

syntax

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's **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's **body**

A plan's body may contain:

- Goal operators (!, ?, !!)
- Belief operators (+, -, -+)
- Actions and Constraints

Example (plan's body)

```
+beer :  now(H) & time_to_leave(T) & H >= T
    <- !g1;           // new sub-goal
        !!g2;         // new goal
    +b1(T-H);        // add new self belief
    -+b2(T*H);       // update belief
    ?b(X);           // new sub-goal
    X > 10;           // constraint to continue the plan
    close(door).     // external action
```

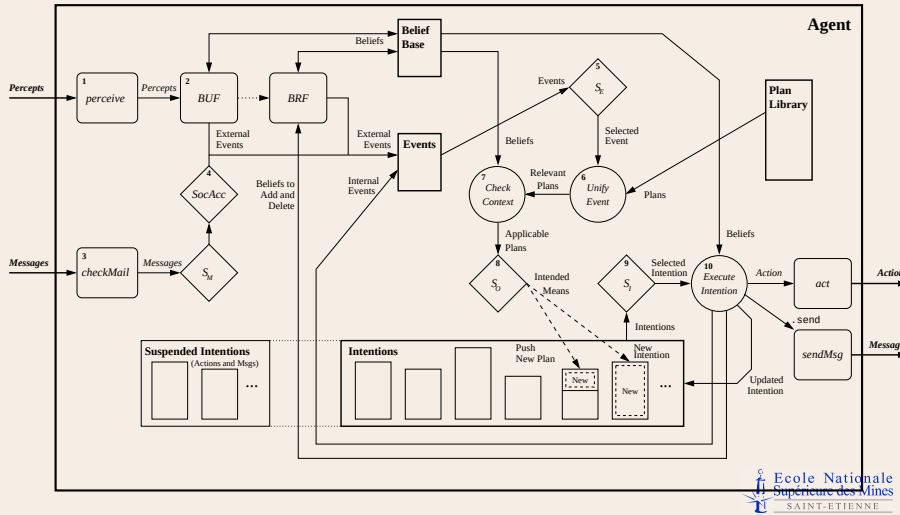
Plans — example

```
+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).
```

Jason basic reasoning cycle

- 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

Jason reasoning cycle



Jason × Java I

Consider a very simple robot with two goals:

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

Jason × Java II

Example (Java code – go to gold)

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

(how to code the charge battery behaviour?)

Jason × Java III

Example (Java code – charge battery)

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

(note where the test for low battery have to be done)

Jason × Java IV

Example (*Jason* code)

```
+see(gold)
  <- !goto(gold).
+!goto(gold) : see(gold)
  <- !select_direction(A);
    go(A);
    !goto(gold).
+battery(low)
  <- .suspend(goto(gold));
    !charge;
    .resume(goto(gold)).
```

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

- 1 Origins and Fundamentals
- 2 Features
 - Strong negation
 - Rules
 - Failure Handling
 - Internal actions
 - Customisations
- 3 Multi-Agent System definition
- 4 Current Shortfalls and Future Trends

Strong negation

Example

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

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

Rules in belief base

Example

```
likely_color(Obj,C) :-  
    colour(Obj,C)[degOfCert(D1)] &  
    not (colour(Obj,_) [degOfCert(D2)] & D2 > D1) &  
    not ~colour(C,B).
```

Failure handling

Example (an agent blindly committed to g)

$+!g : g.$

$+!g : \dots \leftarrow \dots ?g.$

$-!g : \text{true} \leftarrow !g.$

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(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.

Possible customisations in *Jason*

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

- 1 Origins and Fundamentals
- 2 Features
- 3 Multi-Agent System definition
 - Platforms
 - Organisation
 - Environment
 - MAS project
 - Tools
- 4 Current Shortfalls and Future Trends

Communication infrastructure

Different communication and execution management infrastructures can be used with *Jason*:

Centralised: all agents in the same machine,
one thread by agent, very fast

Centralised (pool): all agents in the same machine,
fixed number of thread,
allows thousands of agents

Jade: distributed agents, FIPA-ACL

Saci: distributed agents, KQML

.... others defined by the user (e.g. AgentScape)

$\mathcal{I}$ -MOISE⁺ — organisational programming

- The MAS has an organisation based on roles, groups, global goals, ...

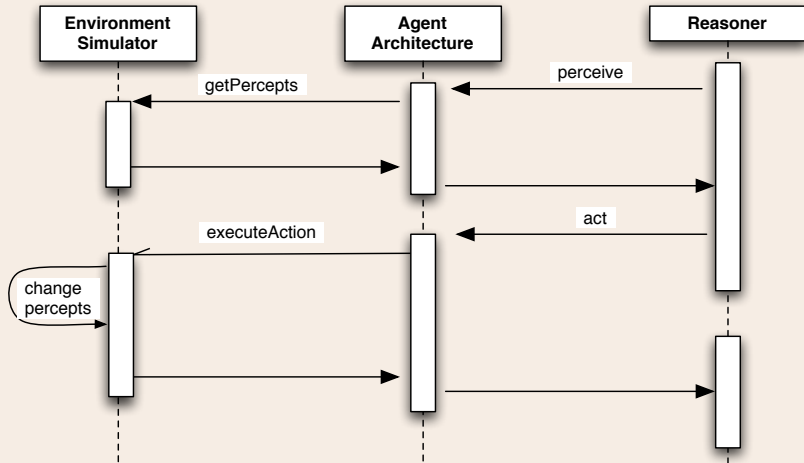
⇒ global orchestration

- Gives *organisational events*
- Allows *organisational action*
- Allows coordination by means of global plans

Definition of a **simulated** environment

- There will normally be an environment where the agents are situated
- The agent architecture needs to be customised to get perceptions and act on such environment
- We often want a simulated environment (e.g. to test a MAS application)
- This is done in Java by extending *Jason's* Environment class

Interaction with the environment simulator



Example of an Environment Class

```
1 import jason.*;
2 import ...;
3 public class robotEnv extends Environment {
4     ....
5     public robotEnv() {
6         Literal gp =
7             Literal.parseLiteral("green_patch(souffle)");
8         addPercept(gp);
9     }
10
11     public boolean executeAction(String ag, Structure action) {
12         if (action.equals(...)) {
13             addPercept(ag,
14                 Literal.parseLiteral("location(souffle,c(3,4))");
15         }
16         ...
17         return true;
18     } }
```

CARTAgO working environments

C4Jason is an integration technology enabling *Jason* agents to operate in cartago

- Extending the set of agent (internal) actions with a basic set to work within artifact-based environments
 - Workspace Management
 - (Artifact) Operation Use
 - Artifact Observation (Observable properties)
 - Artifact instantiation, discovery and management
- Open-source technology
 - available at <http://cartago.sourceforge.net>

MAS configuration language I

- Simple way of defining a multi-agent system

Example (MAS that uses JADE as infrastructure)

```
MAS my_system {  
  infrastructure: Jade  
  environment: robotEnv  
  agents:  
    c3po;  
    r2d2 at jason.sourceforge.net;  
    bob #10; // 10 instances of bob  
  classpath: "../lib/graph.jar";  
}
```

MAS configuration language II

- Configuration of event handling, frequency of perception, user-defined settings, customisations, etc.

Example (MAS with customised agent)

```
MAS custom {  
  agents: bob [verbose=2,paramters="sys.properties"]  
    agentClass MyAg  
    agentArchClass MyAgArch  
    beliefBaseClass jason.bb.JDBCPersistentBB(  
      "org.hsqldb.jdbcDriver",  
      "jdbc:hsqldb:bookstore",  
      ...  
    )  
}
```

MAS configuration language III

Example (CArtAgO environment)

```
MAS grid_world {  
  
    environment: alice.c4jason.CEnv  
  
    agents:  
        cleanerAg  
            agentArchClass alice.c4jason.CogAgentArch  
            #3;  
}
```

jEdit plugin

jEdit - owner.asl

owner.asl (/Users/jomi/Jason/svn-jason/examples/domestic-robot/)

Structure Browser

- owner.asl
 - 3 |!get(beer)
 - 4 |!check_bored
 - 5 |-has(owner,beer)
 - 6 |!drink(beer)
 - 7 |!drink(beer)
 - 8 |!check_bored
 - 9 |msg(M)[source]

DomesticRobot.mas2j owner.asl robot.asl

```

3 |!get(beer). // initial goal: get a beer
4 |!check_bored. // initial goal: verify whether I am getting bored
5
6 |!get(beer) : true
7 |<- .send(robot, achieve, has(owner,beer)).
8
9 |has(owner,beer) : true
10 |<- !drink(beer).
11 |-has(owner,beer) : true
12 |<- !get(beer).
13
14 // while I have beer, sip
15 |!drink(beer) : has(owner,beer)
  
```

about Jason

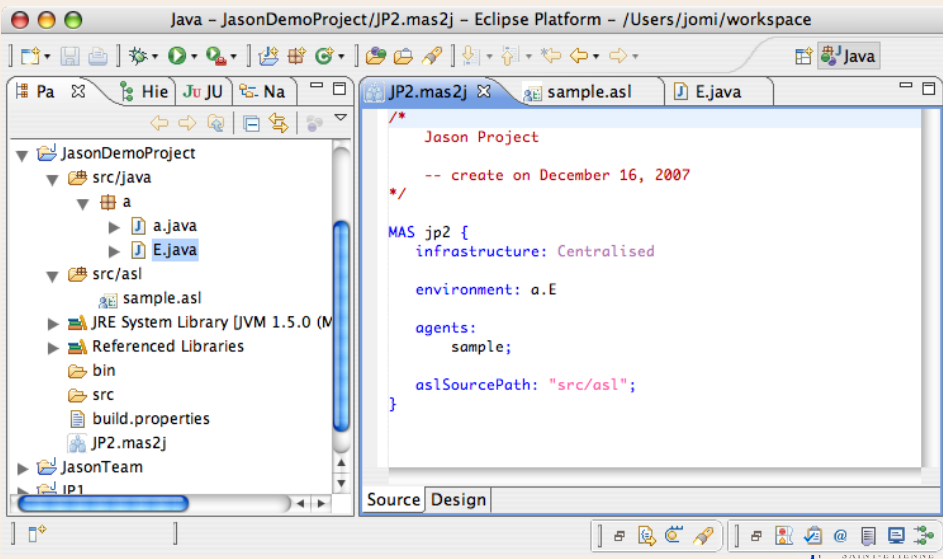
Jason console

Jason functions are also available in the Plugin-Jason menu.

Project agents

robot;
owner;

Eclipse plugin



Mind inspector

Jason Mind Inspector :: cycle 22 ::

Agents

- r2
- r1**

Agent Inspection

Inspection of agent r1 (cycle #12)

- Beliefs

```
pos(back,3,0)[source(self)]·
pos(r1,3,0)[source(percept)]·
pos(r2,3,3)[source(percept)]·
garbage(r1)[source(percept)]·
```

- Events

Sel	Trigger	Intention
X	+!ensure_pick(garb)	4

+ Options

- Intentions

Sel	Id	Pen	Intended Means	Stack (show details)
X	4		+!ensure_pick(S)	{ S = garb }
			+!take(S,L)	{ S = garb, L = r2 }
			+!carry_to(R)	{ R = r2, Y = 0, X = 3 }
			+garbage(r1) _[source(percept)]	

- 1 Origins and Fundamentals
- 2 Features
- 3 Multi-Agent System definition
- 4 Current Shortfalls and Future Trends
 - Projects
 - Shortfalls
 - Trends
 - Summary

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)
- *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)
 - *C*ArtAg*O* integration
 - *M*oise⁺ integration
- More on `jason.sourceforge.net`, related projects

The **State** of Multi-Agent Programming

- Already the *right* way to implement MAS is to use an AOSE Methodology (Prometheus, Gaia, Tropos, ...) and an MAS Programming Language!
- *Jason*, 2APL, Jadex, ... have efficient and stable interpreters — used extensively in teaching
- All have some programming tools (IDE, tracing of agents' mental attitudes, etc.)
- Finally incorporating *social* aspects of MAS
- Growing user base

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 for *Jason* I

- *Modularity* and encapsulation
 - Capabilities (JACK, Jadex, ...)
 - Roles (Dastani et al.)
 - mini-agents (?)
- *Debugging* MAS is hard: problems of concurrency, simulated environments, emergent behaviour, mental attitudes

Some Trends for *Jason* II

- Further work on combining with interaction, environments, and organisations
- We need to put everything together: rational agents, environments, organisations, normative systems, economically inspired techniques, etc.

~> *Multi-Agent Programming*

Summary

- AgentSpeak
 - Logic + BDI
 - Agent programming
- *Jason*
 - AgentSpeak interpreter
 - implements the operational semantics of AgentSpeak
 - speech act based
 - highly customisable
 - useful tools
 - open source
 - open issues

- Many thanks to the
 - various colleagues acknowledged/referenced throughout these slides
 - *Jason* users for useful feedback
- Thanks! Questions?

- Many thanks to the
 - various colleagues acknowledged/referenced throughout these slides
 - *Jason* users for useful feedback
- Thanks! Questions?

—
Whatever you do in MAS, make it available in a
programming language/platform for MAS!!!
—

More information

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

