Multi-Agent Oriented Programming

– Agent-Oriented Programming –

The Jason Agent Programming Language

Olivier Boissier

ENS Mines Saint-Etienne
http://www.emse.fr/~boissier

Ecole Nationale Supérieure des Mines SAINT-ETIENNE

Web Intelligence Master — Nov 2012

Thanks to Jomi F. Hübner, UFSC/DAS Brazil
and Rafael H. Bordini PUCRS Brazil for providing most of the slides

Outline

Agent Oriented Programming

- Use of *mentalistic* notions and a *societal* view of computation
- Various sophisticated abstractions
  - Agent: Belief, Goal, Intention, Plan *(this course)*
  - Organisation: Group, Role, Norm *(see next course)*
  - Interaction: Speech Acts, Interaction protocols *(this course)*
  - Environment: Artifacts, Percepts, Actions *(see next course)*

Agent Oriented Programming
Features

- *Reacting* to events × *long-term* goals
- Course of *actions* depends on *circumstance*
- *Plan failure* (dynamic environments)
- *Rational* behaviour
- *Social* ability
- Combination of *theoretical* and *practical* reasoning

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, ...); SemantiCore (Blois, ...);GOAL (Hindriks); BRAHMS (Sierhuis, ...); 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); ...

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*
- Developed by *Rafael H. Bordini* and *Jomi F. Hübner*

Basics

- As in Prolog, any symbol (i.e. a sequence of characters) starting with a lowercase letter is called an *atom*
- An atom is used to represent particular individuals or objects
- A symbol starting with an uppercase letter is interpreted as a *logical variable*
- Initially variables are *free* or *uninstantiated* and once *instantiated* or *bound* to a particular value, they maintain that value throughout their *scope* (*plan*).
- Variables are bound to values by *unification* ; a formula is called *ground* when it has no more uninstantiated 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
- Events: happen as a consequence to changes in the agent's beliefs or goals
- Intentions: plans instantiated to achieve some goal

Main Architectural Components

- Belief base: where beliefs are stored
- Set of events: to keep track of events the agent will have to handle
- Plan library: stores all the plans currently known by the agent
- Set of Intentions: each intention keeps track of the goals the agent is committed to and the courses of action it chose in order to achieve the goals for one of various foci of attention the agent might have

Jason basic reasoning cycle

- perceives the environment and update belief base
- processes new messages
- selects event
- selects *relevant* plans
- selects *applicable* plans
- creates/updates intention
- selects intention to execute

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 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)]
-+lier(john); // updates 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)
? (test goal)

Example (initial goal of agent Tom)

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


Goals – Dynamics 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);
...
```

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
```


Events – Representation

- Events happen as a 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 (cf. events description)
- the *context* represents 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 to be 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

A plan body may contain:

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

Example (plan's body)

```
+beer :  time_to_leave(T) & clock.now(H) & H >= T
   <- !g1;          // new sub-goal suspending plan execution
      !!g2;         // new goal not suspending plan execution
      +b1(T-H);     // adds new belief
      -+b2(T*H);    // updates belief
      ?b(X);        // new test goal
      X > 10;       // constraint to carry on
      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).
```


Plans – Dynamics

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

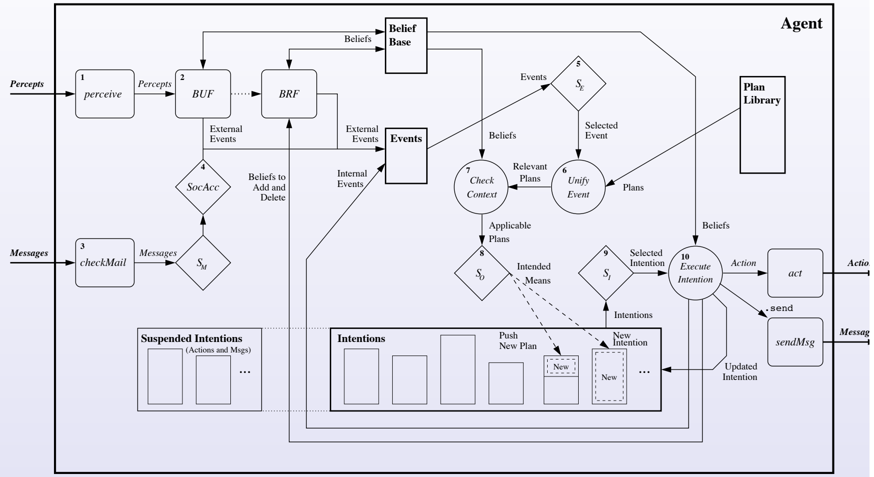
- initial plans defined by the programmer
- plans added dynamically and intentionally by
  - .add_plan
  - .remove_plan
- plans received from messages of type:
  - *tellHow*
  - *untellHow*

  messages

Jason basic reasoning cycle

- perceives the environment and update belief base
- processes new messages
- selects event
- selects *relevant* plans
- selects *applicable* plans
- creates/updates intention
- selects intention to execute

Jason reasoning cycle



Jason vs 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 vs 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 vs 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 vs Java IV

Example (Jason code)

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


Jason vs 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

Negation

Negation as failure

- **not**: formula is true if the interpreter fails to derive it
- *Closed world assumption*: anything that is neither known to be true, nor derivable from the known facts using the rules in the program, is assumed to be false.

Strong negation

- ~: used to express that an agent *explicitly* believes something to be false.

Strong negation

Example

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

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


Prolog-like Rules in the Belief Base

Rules

Rules can be used to simplify certain taks, i.e. making certain conditions used in plans more succinct.
Their syntax is *similar* to the one used for plans.

Example

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


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).
```

Failure handling

Example (an agent blindly committed to g)

```
+!g : g.

+!g : ... <- ... ?g.

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


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

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.

Jason Customisations

- *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, ...)

Execution & Communication Platform

Different execution and communication platforms 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

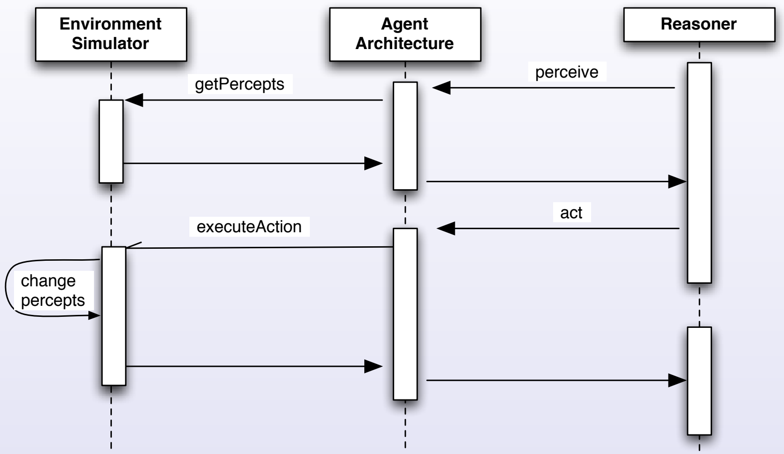
Saci: distributed agents, KQML

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

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 to 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

```
import jason.*;
import ...;
public class robotEnv extends Environment {
  ....
  public robotEnv() {
    Literal gp =
           Literal.parseLiteral("green_patch(souffle)");
    addPercept(gp);
  }

  public boolean executeAction(String ag, Structure action) {
    if (action.equals(...)) {
      addPercept(ag,
           Literal.parseLiteral("location(souffle,c(3,4))");
    }
    ...
    return true;
} }
```


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";
}
```

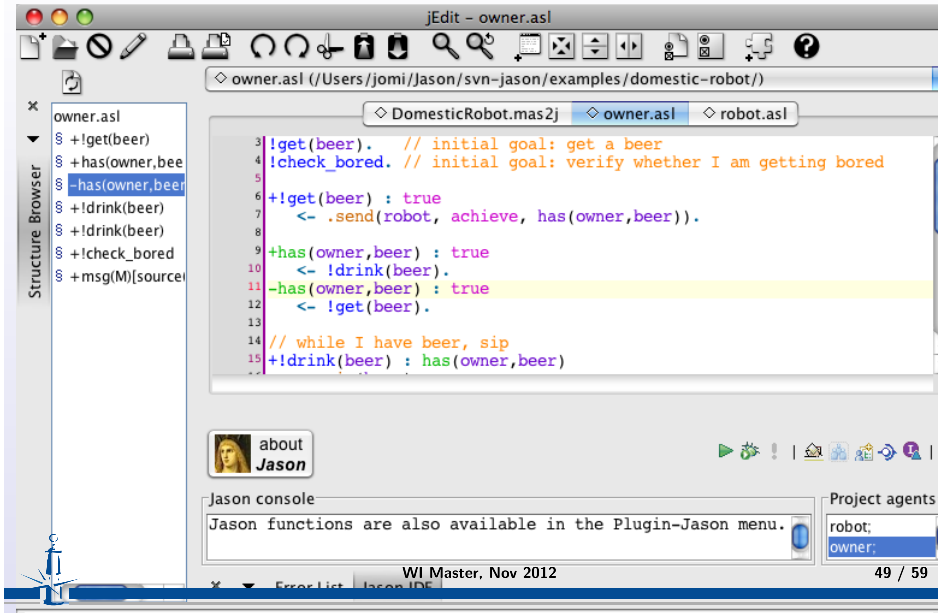

MAS Configuration Language II

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

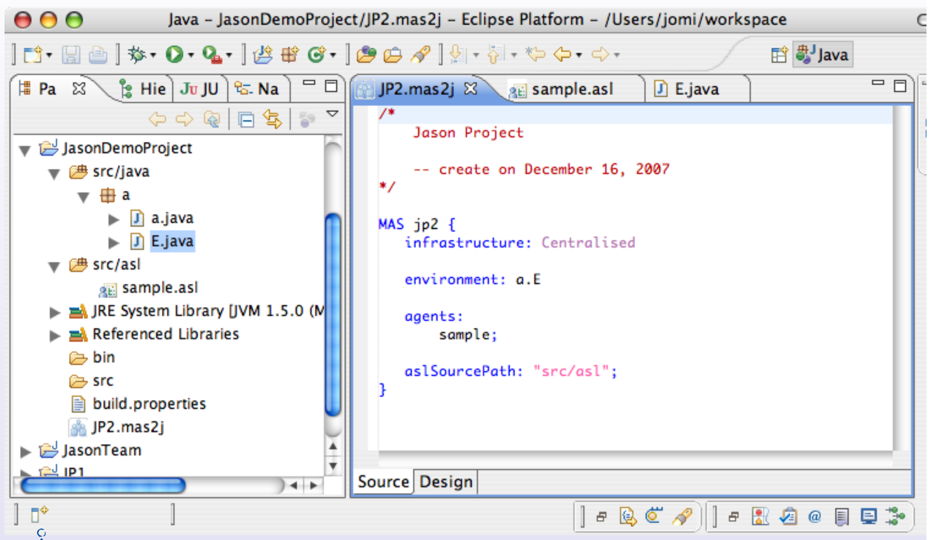
Example (MAS with customised agent)

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

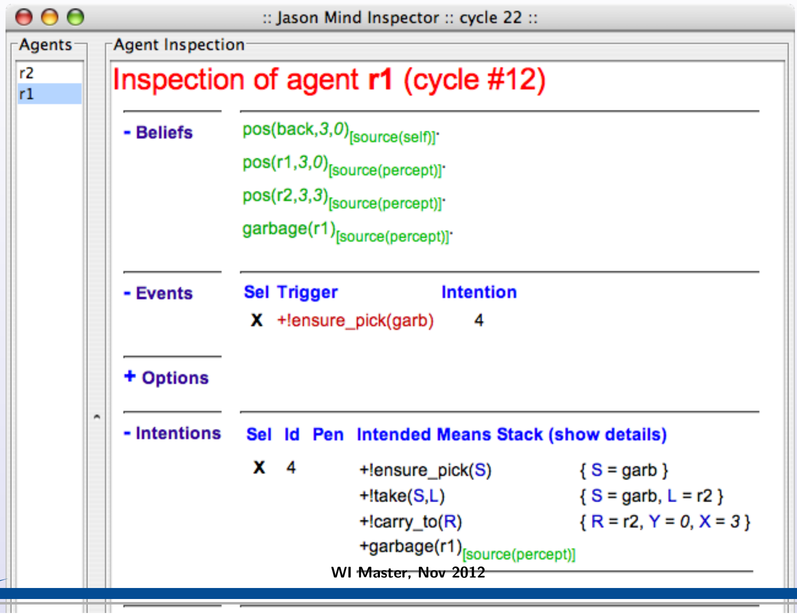
jEdit plugin



Eclipse plugin



Mind inspector



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)
  - *CArtAgO* integration
  - $\mathcal{M}$OISE^{+} integration
- More on `jason.sourceforge.net`, related projects

Some Trends for Jason I

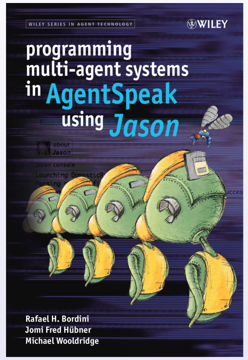
- Modularity and encapsulation
  - Capabilities (JACK, Jadex, ...)
  - Roles (Dastani et al.)
  - Mini-agents (?)
- Recently done: *meta-events*
- To appear soon: annotations for *declarative goals*, improvement in plan failure handling, etc.
- *Debugging* is hard, despite mind inspector, etc.
- Further work on combining with environments and organisations

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

More information

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



Bibliography I

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.