Agent Oriented Programming
with Jason @JaCaMo

Jomi F. Hübner

Federal University of Santa Catarina, Brazil

ALTISSIMO 2014 — Bucharest
Outline of the course

▶ Introduction
▶ AOP – Agent Oriented Programming: Jason
▶ EOP – Environment Oriented Programming: CArtAgO (Alessandro Ricci)
▶ OOP – Organisation Oriented Programming: Moise (Olivier Boissier)
▶ Conclusions
▶ Practical Exercise: a hands-on lab session!
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, ...
A Multi-Agent System (our perspective)

**Definition...**

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, ...

3
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

- ORGANISATION LEVEL
- AGENT LEVEL
- ENDODGESOUS ENVIRONMENT LEVEL
- EXODGESOUS ENVIRONMENT LEVEL

- org
- mission
- role
- agent
- artifact
- wsp
- network node
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 —
Books: [Bordini et al., 2005a], [Bordini et al., 2009]
Proceedings: ProMAS, DALT, LADS, EMAS, ...
Surveys: [Bordini et al., 2006a], [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., 2007b],
- Jadex [Pokahr et al., 2005], 2APL [Dastani, 2008],
- GOAL [Hindriks, 2009], JACK [Winikoff, 2005],
- JIAC, AgentFactory

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); ...
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!
- Many agent languages have efficient and stable interpreters — used extensively in teaching
- All have some programming tools (IDE, tracing of agents’ mental attitudes, tracing of messages exchanged, etc.)
- Finally integrating with social aspects of MAS
- Growing user base
Agent Oriented Programming

Features

- Reacting to events × 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]
BDI architecture
[Wooldridge, 2009]

1 begin
2 while true do
3     p ← perception()
4     B ← brf(B, p) ;
5     D ← options(B, I) ;  // belief revision
6     l ← filter(B, D, I) ;  // desire revision
7     execute(l) ;
8 end
BDI architecture

[Wooldridge, 2009]

1 while true do
2 \[ B \leftarrow \text{brf}(B, \text{perception}()) \]
3 \[ D \leftarrow \text{options}(B, I) \]
4 \[ I \leftarrow \text{filter}(B, D, I) \]
5 \[ \pi \leftarrow \text{plan}(B, I, A) \]
6 while \( \pi \neq \emptyset \) do
7 \[ \text{execute}(\text{head}(\pi)) \]
8 \[ \pi \leftarrow \text{tail}(\pi) \]
BDI architecture
[Wooldridge, 2009]

1 while true do
2 \[ B \leftarrow brf(B, \text{perception}()) \]
3 \[ D \leftarrow \text{options}(B, I) \]
4 \[ I \leftarrow \text{filter}(B, D, I) \]
5 \[ \pi \leftarrow \text{plan}(B, I, A) \]
6 while \( \pi \neq \emptyset \) do
7 \[ \text{execute}(\text{head}(\pi)) \]
8 \[ \pi \leftarrow \text{tail}(\pi) \]
while true do

1. $B \leftarrow \text{brf}(B, \text{perception}())$
2. $D \leftarrow \text{options}(B, I)$
3. $I \leftarrow \text{filter}(B, D, I)$
4. $\pi \leftarrow \text{plan}(B, I, A)$

while $\pi \neq \emptyset$ do

7. execute($\text{head}(\pi)$)
8. $\pi \leftarrow \text{tail}(\pi)$
9. $B \leftarrow \text{brf}(B, \text{perception}())$
10. if $\neg \text{sound}(\pi, I, B)$ then
11. $\pi \leftarrow \text{plan}(B, I, A)$

revise commitment to plan – re-planning for context adaptation
BDI architecture
[Wooldridge, 2009]

while true do
  \( B \leftarrow \text{brf}(B, \text{perception}()) \)
  \( D \leftarrow \text{options}(B, I) \)
  \( I \leftarrow \text{filter}(B, D, I) \)
  \( \pi \leftarrow \text{plan}(B, I, A) \)
  while \( \pi \neq \emptyset \) and \( \neg \text{succeeded}(I, B) \) and \( \neg \text{impossible}(I, B) \) do
    execute( head(\( \pi \)) )
    \( \pi \leftarrow \text{tail}(\pi) \)
    \( B \leftarrow \text{brf}(B, \text{perception}()) \)
  if \( \neg \text{sound}(\pi, I, B) \) then
    \( \pi \leftarrow \text{plan}(B, I, A) \)
  revise commitment to intentions – Single-Minded Commitment
BDI architecture

[Wooldridge, 2009]

1 \textbf{while} true \textbf{do}
2 \hspace{1em} B ← brf (B, perception())
3 \hspace{1em} D ← options(B, I)
4 \hspace{1em} I ← filter(B, D, I)
5 \hspace{1em} π ← plan(B, I, A)
6 \textbf{while} π ≠ {} and ¬succeeded(I, B) and ¬impossible(I, B) \textbf{do}
7 \hspace{2em} execute( head(π) )
8 \hspace{2em} π ← tail(π)
9 \hspace{2em} B ← brf (B, perception())
10 \hspace{2em} \textbf{if} reconsider(I, B) \textbf{then}
11 \hspace{3em} D ← options(B, I)
12 \hspace{3em} I ← filter(B, D, I)
13 \hspace{2em} \textbf{if} ¬sound(π, I, B) \textbf{then}
14 \hspace{3em} π ← plan(B, I, A)

reconsider the intentions (not always!)
Jason

(let's go *programming* those nice concepts)
(BDI) Hello World

i_am(happy).  // B

!say(hello).  // D

+!say(X) : not i_am(sad) <- .print(X).  // I
Desires in Hello World

+i_am(happy) <- !say(hello).

+!say(X) : not i_am(sad) <- .print(X).
Hello World

source of beliefs

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

+!say(X) : not i_am(sad) <- .print(X).
Hello World

plan selection

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

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

+!say(X) : not i_am(sad) <- .print(X).
Hello World

intention revision

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

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

+!say(X) : not i_am(sad) <- .print(X); !say(X).

-is_happy(H)
  : .my_name(H)
  <- .drop_intention(say(hello)).
Hello World

intention revision

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

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

+!say(X) : not i_am(sad) <- .print(X); !say(X).

-is_happy(H)
  : .my_name(H)
  <- .drop_intention(say(hello)).
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
Agent meta-model

main constructors

- Agent
- Belief
- Goal
- Plan
- Trigger event
- Action
  - External Action
  - Internal Action
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**

1. **Percepts**
   - perceive

2. **BUF**
   - External Events
   - Beliefs to Add and Delete

3. **SocAcc**
   - Beliefs

4. **checkMail**
   - Messages

5. **BRF**
   - BRF

6. **Events**
   - External Events
   - Internal Events

7. **Check Context**
   - Relevant Plans

8. **Suspended Intentions**
   - (Actions and Msgs)

9. **Unify Event**
   - Plans

10. **Execute Intention**
    - Action

**Beliefs Base**

- New Plan
- Push New Plan
- New Intention
- Selected Intention
- Intended Means
- Selected Event
- Applicable Plans
- Events
- Relevant Plans
- Plans
- New Plans
- New Intention
- Suspended Intentions

**Percepts**
- Messages
- Actions

**Actions**
- sendMsg
- send
- Messages
Beliefs — Representation

**Syntax**

Beliefs are represented by annotated literals of first order logic

\[
\text{functor}(\text{term}_1, \ldots, \text{term}_n)[\text{annot}_1, \ldots, \text{annot}_m]
\]

**Example (belief base of agent Tom)**

\[
\begin{align*}
\text{red(box1)}[\text{source(percept)}]. \\
\text{friend(bob,alice)}[\text{source(bob)}]. \\
\text{lier(alice)}[\text{source(self),source(bob)}]. \\
\sim\text{lier(bob)}[\text{source(self)}].
\end{align*}
\]
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)]`
by communication

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

```plaintext
.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).
by intention

the plan operators `!` and `?` can be used to add a new goal annotated with `source(self)`

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

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

Goals — Dynamics I
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
```
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

\[ \text{send(tom, askOne, published(P), Answer); // sent by Bob} \]
// adds new goal \(?\text{publisher(P)[source(bob)]}\) 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)
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).
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
Other Language Features

Strong Negation

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

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

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

Example (an agent blindly committed to g)

+!g : g.

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

-!g : true <- !g.
Meta Programming

Example (an agent that asks for plans \textit{on demand})

\[-!G[\text{error(no\_relevant)}] : \text{teacher}(T)\]
\[\leftarrow \text{send}(T, \text{askHow}, \{ \text{+!G} \}, \text{Plans});\]
\[\text{add\_plan}(\text{Plans});\]
\[!G.\]

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

- The failure event is annotated with the error type, line, source, ... \texttt{error(no\_relevant)} means no plan in the agent’s plan library to achieve \texttt{G}
- \{ \text{+!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($term_1, term_2, \ldots$)
  - .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.
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
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));
            }
        }
    }
}

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();
            }
        }
    }
}

Example (*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)).
```
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
Various 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

... others defined by the user (e.g. AgentScape)
Simple way of defining a multi-agent system

Example (MAS that uses JADE as infrastructure)

```plaintext
mas my_system {
    agent c3po
    agent r2d2
    agent bob
    workspace robots { .... }
}
mas-deployment {
    ag-instances: 1 c3po
    1 r2d2 @ 10.10.1.2
    10 bob
    ws-instances: robots
    platform: jade()
}
```
Configuration of event handling, frequency of perception, user-defined settings, customisations, etc.

Example (MAS with customised agent)

```java
agent bob {
    verbose: 2
    ag-class: MyAg
    ag-arch: MyAgArch
    ag-bb-class: jason.bb.JDBCPersistentBB(
        "org.hsqldb.jdbcDriver",
        "jdbc:hsqldb:bookstore",
        ...
    }
```
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 available with *Jason*: persistent belief base (in text files, in data bases, ...)
Tools

▶ Eclipse Plugin
▶ Mind Inspector
▶ Integration with
  – CArtAgO
  – Moise
  – MADEM
  – Ontologies
  – ...

▶ More on http://jason.sourceforge.net/wp/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
Acknowledgements

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

▶ http://jason.sourceforge.net

▶ R.H. Bordini, J.F. Hübner, and M. Wooldrige
Programming Multi-Agent Systems in AgentSpeak using Jason
Bibliography I


Baldoni, M., Son, T. C., van Riemsdijk, M. B., and Winikoff, M., editors (2008).
Bibliography II

Baldoni, M., Son, T. C., van Riemsdijk, M. B., and Winikoff, M., editors (2009).


Bibliography VII


Jadex: A bdi reasoning engine.
In [Bordini et al., 2005a], pages 149–174.

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

Agent-oriented programming.

Jack intelligent agents: An industrial strength platform.
In [Bordini et al., 2005a], pages 175–193.
An Introduction to MultiAgent Systems.