Multi-Agent Oriented Programming using JaCaMo

Jomi F. Hübner

Federal University of Santa Catarina, Brazil

Summer School on AI for Industry 4.0
2021
From knowledge to action

From theoretical to *practical* reasoning

From mind to body & environment & others (interaction)

From individuals to *societies*
An MAS is a loosely coupled network of problem solvers that interact to solve problems that are beyond the individual capabilities or knowledge of each problem solver – Durfee and Lesser 1989
Outline

- Agents
  - Practical reasoning with JaCaMo
- Environment
- Organisation
- MAOP

(slides written together with R. Bordini, O. Boissier, and A. Ricci)
Agent Oriented Programming — AOP —

- Olivier Boissier, Rafael H. Bordini, Jomi Hübner and Alessandro Ricci
  *Multi-Agent Oriented Programming: Programming Multi-Agent Systems Using JaCaMo*

- (Chapter 6 for this part)
Develop a system to control the temperature of a room so that it reaches some desired temperature

▶ The room is equipped with a “Heating, Ventilating and Air Conditioning” (HVAC)
▶ The HVAC has an MQTT interface that provides
  ▶ the current room temperature
  ▶ operations to start cooling, start heating, and stop the machine
Develop a system to control the temperature of a room so that it reaches some desired temperature

- The room is equipped with a “Heating, Ventilating and Air Conditioning” (HVAC)
- The HVAC has an MQTT interface that provides
  - the current room temperature
  - operations to start cooling, start heating, and stop the machine
Smart Room Scenario

approach

Separation of concerns

- integration and interoperability with the HVAC
- strategy to keep the right temperature
Separation of concerns

- integration and interoperability with the HVAC
  ~ environment modelling
- strategy to keep the right temperature
  ~ agent modelling
Smart Room Scenario

general view

 MAS

agents endogenous environment

agent

artifact

exogenous environment

artifact

exogenous environment
Smart Room Scenario

artifact

rc: room_controller

hvac : HVAC

startCooling/0
startHeating/0
stopAirConditioner/0

temperature/1
state/1
Agent and Environment Integration

▶ (artifact) observable properties $\rightsquigarrow$ (agent) beliefs

▶ (agent) actions $\rightsquigarrow$ (artifact) operations
Agent and Environment Integration

- (artifact) observable properties $\rightsquigarrow$ (agent) beliefs

```csharp
hvac::
  state("idle")[artifact_id(obj_3), artifact_name(hvac), percept_type(obs_prop), source(percept), works
  temperature(41)][artifact_id(obj_3), artifact_name(hvac), percept_type(obs_prop), source(percept),
```

- (agent) actions $\rightsquigarrow$ (artifact) operations

```csharp
!act. // initial goal
+!act <- startCooling. // plan
```
Reactive Behaviour

An agent can react to changes in its mental state:

- new goals

- new beliefs
Reactive Behaviour

An agent can react to changes in its mental state:

- **new goals**
  
  ```
  +!act // new goal
  <- startCooling. // action
  ```

- **new beliefs**
  
  ```
  +temperature(X) // new belief
  : X < 0 // context for the plan
  <- stopAirConditioner. // action
  ```
Choosing actions based on the context

\[
\begin{align*}
\text{temperature}(27). \\
+\text{temperature}(T) : \text{temperature}(C) \ & \ C > T \ & \ - \text{startCooling}. \\
+\text{temperature}(T) : \text{temperature}(C) \ & \ C < T \ & \ - \text{startHeating}. \\
+\text{temperature}(T). \ // \ nothing \ to \ do
\end{align*}
\]

\[
\begin{align*}
+\text{temperature}(T) : \ T < 0 \ & \ - \text{stopAirConditioner}. \\
+\text{temperature}(T) : \ T > 40 \ & \ - \text{stopAirConditioner}.
\end{align*}
\]
Room Controller Agent
version 1 — reactive agent

preference(20).

+preference(P) <- !temperature(P).

+!temperature(T) : hvac::temperature(C) & C > T <- startCooling.
+!temperature(T) : hvac::temperature(C) & C < T <- startHeating.
+!temperature(T). // nothing to do

+hvac::temperature(T) : preference(P) & T == P <- stopAirConditioner.
Room Controller Agent
version 1 — reactive agent

preference(20).

+preference(P) <- !temperature(P).

+!temperature(T) : hvac::temperature(C) & C > T <- startCooling.
+!temperature(T) : hvac::temperature(C) & C < T <- startHeating.
+!temperature(T). // nothing to do

+hvac::temperature(T) : preference(T) <- stopAirConditioner.
Room Controller Agent
version 1 — reactive agent

preference(20).

+preference(P) <- !temperature(P).

+!temperature(T) : hvac::temperature(C) & C > T <- startCooling.
+!temperature(T) : hvac::temperature(C) & C < T <- startHeating.
+!temperature(T). // nothing to do

+hvac::temperature(T) : preference(T) <- stopAirConditioner.

what happens if the current temperature changes?
Room Controller Agent
version 1 — reactive agent

preference(20).

+preference(P) <- !temperature(P).

+!temperature(T) : hvac::temperature(C) & C > T <- startCooling.
+!temperature(T) : hvac::temperature(C) & C < T <- startHeating.
+!temperature(T). // nothing to do

+hvac::temperature(T) : preference(T) <- stopAirConditioner.
+hvac::temperature(T) : preference(P) & T \leq P <- !temperature(P).
Room Controller Agent
version 1 — reactive agent

preference(20).

+preference(P) <- !temperature(P).

+!temperature(T) : hvac::temperature(C) & C > T <- startCooling.
+!temperature(T) : hvac::temperature(C) & C < T <- startHeating.
+!temperature(T).  // nothing to do

+hvac::temperature(T) : preference(T) <- stopAirConditioner.
+hvac::temperature(T) : preference(P) <- !temperature(P).
Room Controller Agent
version 1 — reactive agent

preference(20).

+preference(P) <- !temperature(P).

+!temperature(T) : hvac::temperature(C) & C > T <- startCooling.
+!temperature(T) : hvac::temperature(C) & C < T <- startHeating.
+!temperature(T). // nothing to do

+hvac::temperature(T) : preference(T) <- stopAirConditioner.
+hvac::temperature(T) : preference(P) <- !temperature(P).

what happens if the preference changes?
Room Controller Agent
version 1 — reactive agent

preference(20).

+preference(P) <- !temperature(P).

+!temperature(T) : hvac::temperature(C) & C > T <- startCooling.
+!temperature(T) : hvac::temperature(C) & C < T <- startHeating.
+!temperature(T). // nothing to do

+hvac::temperature(T) : preference(T) <- stopAirConditioner.
+hvac::temperature(T) : preference(P) <- !temperature(P).

what happens if the preference changes?

we end up creating many (reactive) plans that create the goal.
Room Controller Agent
version 2 — pro-active agent – **maintenance goal**

\[\text{preference}(20).\]

\[!\text{start}.\]
\[+!\text{start} : \text{preference}(P) \leftarrow !\text{keep\_temperature}.\]

\[+!\text{keep\_temperature}\]
\[\quad : \text{preference}(P) \& \text{hvac::temperature}(C) \& C > P\]
\[\quad \leftarrow \text{startCooling}; !\text{wait\_until}(P); !\text{keep\_temperature}.\]

\[+!\text{keep\_temperature}\]
\[\quad : \text{preference}(P) \& \text{hvac::temperature}(C) \& C < P\]
\[\quad \leftarrow \text{startHeating}; !\text{wait\_until}(P); !\text{keep\_temperature}.\]

\[+!\text{keep\_temperature}\]
\[\quad \leftarrow \text{wait}(100); !\text{keep\_temperature}.\]

\[+!\text{wait\_until}(T) : \text{hvac::temperature}(T) \leftarrow \text{stopAirConditioner}.\]
\[+!\text{wait\_until}(T) \leftarrow !\text{wait\_until}(T). // \text{busy waiting (!)}\]
Room Controller Agent
version 2 — pro-active agent – maintenance goal

preference(20).

!start.
+!start : preference(P) <- !keep_temperature.

+!keep_temperature
   : preference(P) & hvac::temperature(C) & C > P
   <- startCooling; !wait_until(P); !keep_temperature.
+!keep_temperature
   : preference(P) & hvac::temperature(C) & C < P
   <- startHeating; !wait_until(P); !keep_temperature.
+!keep_temperature
   <- .wait(100); !keep_temperature.

+!wait_until(T) : hvac::temperature(T) <- stopAirConditioner.
+!wait_until(T) <- !wait_until(T). // busy waiting (!)

does it react to changes in the temperature? What is the delay?
Room Controller Agent
version 3 — no busy waiting

preference(20).

!start.
+!start : preference(P) <= !keep_temperature.

+!keep_temperature
  : preference(P) & hvac::temperature(C) & C > P
  <= startCooling; !wait_until(P); !keep_temperature.
+!keep_temperature
  : preference(P) & hvac::temperature(C) & C < P
  <= startHeating; !wait_until(P); !keep_temperature.
+!keep_temperature
  <= .wait( {+hvac::temperature(_)}, 100, _); !keep_temperature.

+!wait_until(T) : hvac::temperature(T) <= stopAirConditioner.
+!wait_until(T) <= .wait( {+hvac::temperature(_)} ); !wait_until(T).
preference(20).

!start.
+!start : preference(P) <- !keep_temperature.

+!keep_temperature
  : preference(P) & hvac::temperature(C) & C > P
  <- startCooling; !wait_until(P); !keep_temperature.
+!keep_temperature
  : preference(P) & hvac::temperature(C) & C < P
  <- startHeating; !wait_until(P); !keep_temperature.
+!keep_temperature
  <- .wait( {+hvac::temperature(_)}, 100, _); !keep_temperature.

+!wait_until(T) : hvac::temperature(T) <- stopAirConditioner.
+!wait_until(T) <- .wait( {+hvac::temperature(_)} ); !wait_until(T).

does it react to changes in the preference? What is the delay?
... 

hot  :-  preference(P) & hvac::temperature(C) & C > P.
cold :-  preference(P) & hvac::temperature(C) & C < P.

+!keep_temperature
  :  hot
    <- startCooling; !wait_until_pref; !keep_temperature.
+!keep_temperature
  :  cold
    <- startHeating; !wait_until_pref; !keep_temperature.

...
Agents

- promptly react to the environment
- have goal oriented behaviour
- select actions based on circumstance
- combines theoretical and practical reasoning
Fundamentals
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); MedateM (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 (Torrioni, 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
Fundamentals

- Use of *mentalistic* notions and a *societal* view of computation [Shoham, 1993]

- Heavily influenced by the BDI architecture and reactive planning systems [Bratman et al., 1988]
BDI architecture
(the mentalistic view)
AgentSpeak
The foundational language for *Jason*

- Programming language for BDI agents
- Originally proposed by Rao [Rao, 1996]
- 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 — agent dimension in JaCaMo

- *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
Beliefs — Representation

Syntax

Beliefs are represented by annotated literals of first order logic

\[
\text{functor}(term_1, \ldots, term_n)[annot_1, \ldots, annot_m]
\]

Example (belief base of agent Tom)

\[
\text{red(box1)}[\text{source(percept)}]. \\
\text{friend(bob,alice)}[\text{source(bob)}]. \\
\text{lier(alice)}[\text{source(self)},\text{source(bob)}]. \\
\neg \text{lier(bob)}[\text{source(self)}].
\]
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);

...```

```plaintext

...```
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

```lang
.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

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

\[
\text{triggering\_event \ : \ context \ \leftarrow \ \text{body}}.
\]

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)
\textbf{rain} : time\_to\_leave(T) \& clock\_now(H) \& H \geq T

\(-gi; \quad // \text{new sub-goal}
\)

\(-g2; \quad // \text{new goal}
\)

?b(X); \quad // \text{new test goal}

+b1(T-H); \quad // \text{add mental note}

-b2(T-H); \quad // \text{remove mental note}

++b3(T*H); \quad // \text{update mental note}

\text{jia.get}(X); \quad // \text{internal action}

X > 10; \quad // \text{constraint to carry on}

\text{close(door);} \quad // \text{external action}

\text{!g3[hard\_deadline(3000)]}. \quad // \text{goal with deadline}
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`
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

1. Percepts
   - perceive
   - BUF
   - BRF

2. External Events
   - Beliefs to Add and Delete
   - Selected

3. Messages
   - checkMail
   - SM

4. SocAcc
   - Beliefs to Add and Delete

5. Events
   - Relevant Plans
   - Beliefs
   - Selected

6. Unify Event
   - Plans
   - Beliefs

7. Check Context
   - Relevant Plans
   - Beliefs
   - Internal Events

8. Intended Means
   - Intention
   - Selected

9. Selected Intention
   - Intention
   - New
   - Suspended Intentions
   - New

10. Execute Intention
    - Action
    - act
    - send
    - sendMsg

Suspended Intentions
   - (Actions and Msgs)

Intentions
   - Push
   - New Plan

Updated Intention

Plan Library

Actions

Messages
Jason Reasoning Cycle

- **Percepts**
  - 1. perceive
  - 2. BUF
  - 4. SocAcc

- **Messages**
  - 3. checkMail
  - 4. SM

- **External Events**

- **Actions and Messages**
  - send
  - SocAcc
  - 4. Messages

- **Areas of Research**
  - ▶ machine perception
  - ▶ belief revision
  - ▶ 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
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 Features
Example (an agent blindly committed to g)

```prolog
+!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)

```prolog
+!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).
+\text{f} : .super_goal(g,SG) <- .fail_goal(SG).
\text{f} is the drop condition for goal \text{g}
```
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)

```plaintext
-!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

```prolog
+!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.
tall(X) :- man(X) & height(X, H) & H > 1.80.
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:

  ```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.
Namespaces & Modularity
Namespaces & Modularity

**Inspection of agent alice**

---

**- Beliefs**

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

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

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

```
tv::
introduction(participant)[source(comparator)]
propose(11.075337225252543)[source]
propose(12.043311087442898)[source]
propose(12.8127904935436)[source]
winner(company_A1)[source(self)].

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

pc::
introduction(participant)[source(comparator)]
propose(11.38950048463455)[source]
propose(11.392553683771682)[source]
propose(12.34890100262853)[source]
winner(company_A2)[source(self)].
```
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, ...)
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();
            }
        }
    }
}
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(executing)]  // 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
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
Further Resources

▶ http://jason.sourceforge.net

▶ R.H. Bordini, J.F. Hübner, and M. Wooldrige

**Programming Multi-Agent Systems in AgentSpeak using Jason**

Environment Oriented Programming — EOP —
?
The notion of environment is intrinsically related to the notion of agent and multi-agent system

- “An agent is a computer system that is situated in some environment and that is capable of autonomous action in this environment in order to meet its design objective” [Wooldridge, 2002]
- “An agent is anything that can be viewed as perceiving its environment through sensors and acting upon the environment through effectors.” [Russell and Norvig, 2003]

Including both physical and software environments
Perception

- process inside agent inside of attaining awareness or understanding sensory information, creating percepts perceived form of external stimuli or their absence

Actions

- the means to affect, change or inspect the environment
Multi-Agent Perspective

In evidence

- overlapping spheres of visibility and influence
- ...which means: interaction
Why Environment Programming

▶ Basic level
▶ to create testbeds for real/external environments
▶ to ease the interface/interaction with existing software environments

▶ Advanced level
▶ to uniformly encapsulate and modularise functionalities of the MAS out of the agents
▶ typically related to interaction, coordination, organisation, security
▶ externalisation
▶ this implies changing the perspective on the environment
▶ environment as a first-class abstraction of the MAS
▶ endogenous environments (vs. exogenous ones)
▶ programmable environments
Basic Level Overview

MAS

actions

SIMULATED WORLD

OR

INTERFACE

OR

WRAPPER TO EXISTING TECHNOLOGY

REAL WORLD
(PHYSICAL OR COMPUTATIONAL)

EXTERNAL WORLD
(PHYSICAL OR COMPUTATIONAL)

Example: JAVA PLATFORM

EXTERNAL WORLD
(PHYSICAL OR COMPUTATIONAL)

REAL WORLD
(PHYSICAL OR COMPUTATIONAL)

mas ENVIRONMENT

actions

percepts
Advanced Level Overview [Weyns et al., 2007]
Existing Computational Frameworks

- **AGRE / AGREEN / MASQ [Stratulat et al., 2009]**
  - AGRE – integrating the AGR (Agent-Group-Role) organisation model with a notion of environment
    - Environment used to represent both the physical and social part of interaction
  - AGREEN / MASQ – extending AGRE towards a unified representation for physical, social and institutional environments
  - Based on MadKit platform [Gutknecht and Ferber, 2000]

- **GOLEM [Bromuri and Stathis, 2008]**
  - Logic-based framework to represent environments for situated cognitive agents
  - composite structure containing the interaction between cognitive agents and objects

- **A&A and CArtAgO [Ricci et al., 2010a]**
  - introducing a computational notion of artifact to design and implement agent environments
A&A and CArtAgO
Agents and Artifacts (A&A) Conceptual Model: Background Human Metaphor
A&A Meta-Model in More Detail [Ricci et al., 2010a]
A World of Artifacts

- A counter
  - count 5
  - inc
  - reset

- A flag
  - state true
  - switch

- A Stock Quote Web Service
  - state available
  - wsdl ...
  - GetLastTradePrice
  - ...

- A data-base
  - n_records 1001
  - table_names ...
  - ...
  - createTable
  - addRecord
  - query
  - ...

- A bounded buffer
  - n_items 0
  - max_items 100
  - put
  - get

- An agenda
  - next_todo check_plant
  - last_todo ...
  - setTodo
cancelTodo

- An event service
  - clearEvents
  - postEvent
  - registerForEvs

- A tuple space
  - out
  - in
  - rd
Actions and Percepts in Artifact-Based Environments [Ricci et al., 2010b]

actions $\longleftrightarrow$ artifacts’ operation

the action repertoire is given by the dynamic set of operations provided by the overall set of artifacts available in the workspace can be changed by creating/disposing artifacts

▶ action success/failure semantics is defined by operation semantics

percepts $\longleftrightarrow$ artifacts’ observable properties $+$ signals

properties represent percepts about the state of the environment

signals represent percepts concerning events signalled by the environment
Interaction Model: Use

- Performing an action corresponds to triggering the execution of an operation
  - acting on artifact’s usage interface
Interaction Model: Operation execution

- a process structured in one or multiple transactional steps
- asynchronous with respect to agent
  - ...which can proceed possibly reacting to percepts and executing actions of other plans/activities
- operation completion causes action completion
  - action completion events with success or failure, possibly with action feedbacks
Agents can dynamically select which artifacts to observe
  - predefined focus/stopFocus actions
By focussing an artifact
- observable properties are mapped into agent dynamic knowledge about the state of the world, as percepts
  - e.g. belief base
- signals are mapped as percepts related to observable events
CArtAgO

- Common ARtifact infrastructure for AGent Open environment (CArtAgO) [Ricci et al., 2009a]
- Computational framework / infrastructure to implement and run artifact-based environment [Ricci et al., 2007]
  - Java-based programming model for defining artifacts
  - set of basic API for agent platforms to work within artifact-based environment
- Distributed and open MAS
  - workspaces distributed on Internet nodes
    - agents can join and work in multiple workspace at a time
  - Role-Based Access Control (RBAC) security model
- Open-source technology
  - available at https://github.com/CArtAgO-lang/cartago
Example 1: A Simple Counter Artifact

class Counter extends Artifact {

  void init(){
    defineObsProp("count",0);
  }

  @OPERATION void inc(){
    ObsProperty p = getObsProperty("count");
    p.updateValue(p.intValue() + 1);
    signal("tick");
  }
}

Some API spots

- Artifact base class
- @OPERATION annotation to mark artifact’s operations
- set of primitives to work define/update/.. observable properties
- signal primitive to generate signals
Example 1: User and Observer Agents

USER(S)

!create_and_use.

+!create_and_use : true
  <- !setupTool(Id);
     // use
     inc;
     // second use specifying the Id
     inc [artifact_id(Id)].

// create the tool
+!setupTool(C): true
  <- makeArtifact("c0","Counter",C).

OBSERVER(S)

!observe.

+!observe : true
  <- ?myTool(C);  // discover the tool
     focus(C).

+count(V)
  <- println("observed new value: ",V).

+tick [artifact_name(Id,"c0")]
  <- println("perceived a tick").

+?myTool(CounterId): true
  <- lookupArtifact("c0",CounterId).

-?myTool(CounterId): true
  <- .wait(10);
     ?myTool(CounterId).

▶ Working with the shared counter
Given the action/operation map, by executing an action the intention/activity is suspended until the corresponding operation has completed or failed.

- action completion events generated by the environment and automatically processed by the agent/environment platform bridge
- no need of explicit observation and reasoning by agents to know if an action succeeded

However, the agent execution cycle is not blocked!

- the agent can continue to process percepts and possibly execute actions of other intentions
Summary

▶ Environment programming
  ▶ environment as a programmable part of the MAS
  ▶ encapsulating and modularising functionalities useful for agents’ work

▶ Artifact-based environments
  ▶ artifacts as first-class abstraction to design and program complex software environments
    ▶ usage interface, observable properties / events, linkability
  ▶ artifacts as first-order entities for agents
    ▶ interaction based on use and observation
    ▶ agents dynamically co-constructing, evolving, adapting their world

▶ CArtAgO computational framework
  ▶ programming and executing artifact-based environments
  ▶ integration with heterogeneous agent platforms
Organisation Oriented Programming — OOP —
Introduction: Some definitions

- Organisations are structured, patterned systems of activity, knowledge, culture, memory, history, and capabilities that are distinct from any single agent [Gasser, 2001]  
  ⇒ organisations are **supra-individual** phenomena

- A decision and communication schema which is applied to a set of actors that together fulfill a set of tasks in order to satisfy goals while guarantying a global coherent state [Malone, 1999]  
  ⇒ definition by the designer, or by actors, to achieve a **purpose**

- An organisation is characterised by: a division of tasks, a distribution of roles, authority systems, communication systems, contribution-retribution systems [Bernoux, 1985]  
  ⇒ **pattern of predefined cooperation**

- An arrangement of relationships between components, which results into an entity, a system, that has unknown skills at the level of the individuals [Morin, 1977]  
  ⇒ **pattern of emergent cooperation**
Organisation in MAS – a definition

- Pattern of agent \textit{cooperation}
  - with a purpose
  - supra-agent
  - emergent or
  - predefined (by designer or agents)
Organisation Oriented Programming (OOP)

- Programming outside the agents
- Using organisational concepts
- To define a cooperative pattern
- Program = Specification
- By changing the specification, we can change the MAS overall behaviour
First approach
- Agents read the program and follow it
Second approach

- Agents **are forced** to follow the program
- Agents **are rewarded** if they follow the program
- ...

Organisation Oriented Programming (OOP)
Second approach

- Agents **are forced** to follow the program
- Agents **are rewarded** if they follow the program
- ...
Organisation Oriented Programming (OOP)

Components
- Programming language (OML)
- Platform (OMI)
- Integration to agent architectures and to environment
Some OOP approaches

- AGR/Madkit [Ferber and Gutknecht, 1998]
- STEAM/Teamcore [Tambe, 1997]
- ISLANDER/AMELI [Esteva et al., 2004]
- Opera/Operetta [Dignum and Aldewereld, 2010]
- PopOrg [Rocha Costa and Dimuro, 2009]
- 2OPL [Dastani et al., 2009]
- THOMAS [Criado et al., 2011],
- ...
Moise Framework

- OML (language)
  - Tag-based language
    (issued from Moise [Hannoun et al., 2000],
    Moise\(^+\) [Hübner et al., 2002],
    Moiselnst [Gâteau et al., 2005])

- OMI (infrastructure)
  - developed as an artifact-based working environment
    (ORA4MAS [Hübner et al., 2009] based on CArtAgO nodes,
    refactoring of S-MOISE\(^+\) [Hübner et al., 2006] and
    SYNAI [Gâteau et al., 2005])

- Integrations
  - Agents and Environment (c4Jason, c4Jadex
    [Ricci et al., 2009b])
  - Environment and Organisation ([Piunti et al., 2009])
  - Agents and Organisation (J-Moise\(^+\) [Hübner et al., 2007])
Moise OML meta-model (partial view)
Moise OML

- OML for defining organisation specification and organisation entity
- Three independent dimensions [Hübner et al., 2007] (well adapted for the reorganisation concerns):
  - **Structural**: Roles, Groups
  - **Functional**: Goals, Missions, Schemes
  - **Normative**: Norms (obligations, permissions, interdictions)
- Abstract description of the organisation for
  - the designers
  - the agents
    - $J$-Moise$^+$ [Hübner et al., 2007]
  - the Organisation Management Infrastructure
    - ORA4MAS [Hübner et al., 2009]
Structural Specification

- Specifies the structure of an MAS along three levels:
  - **Individual** with **Role**
  - **Social** with **Link**
  - **Collective** with **Group**

- Components:
  - **Role**: label used to assign rights and constraints on the behavior of agents playing it
  - **Link**: relation between roles that directly constrains the agents in their interaction with the other agents playing the corresponding roles
  - **Group**: set of links, roles, compatibility relations used to define a shared context for agents playing roles in it
Structural Specification Example

Graphical representation of structural specification of 3-5-2 Joj Team
Functional Specification

- Specifies the expected behaviour of an MAS in terms of **goals** along two levels:
  - **Collective** with **Scheme**
  - **Individual** with **Mission**

- Components:
  - **Goals:**
    - **Achievement goal** (default type). Goals of this type should be declared as satisfied by the agents committed to them, when achieved
    - **Maintenance goal**. Goals of this type are not satisfied at a precise moment but are pursued while the scheme is running. The agents committed to them do not need to declare that they are satisfied
  - **Scheme**: global goal decomposition tree assigned to a group
    - Any scheme has a root goal that is decomposed into subgoals
  - **Missions**: set of coherent goals assigned to roles within norms
score a goal

- $m_1$
  - get the ball
  - $m_1$
    - go towards the opponent field
    - $m_2$
      - be placed in the middle field
    - $m_3$
      - be placed in the opponent goal area
  - $m_2$
    - kick the ball to the goal area
    - go to the opponent back line
  - $m_3$
    - shot at the opponent’s goal
    - kick the ball to (agent committed to $m_2$)

**Key**

- Scheme
- missions
- goal
- success rate
- sequence
- choice
- parallelism

**Organizational Entity**

- Lucio — $m_1$
- Cafu — $m_2$
- Rivaldo — $m_3$

Graphical representation of social scheme “side_attack” for joj team
Normative Specification

- Explicit relation between the functional and structural specifications
- Permissions and obligations to commit to missions in the context of a role
- Makes explicit the normative dimension of a role
### Norm Specification – example

<table>
<thead>
<tr>
<th>role</th>
<th>deontic</th>
<th>mission</th>
<th>TTF</th>
</tr>
</thead>
<tbody>
<tr>
<td>back</td>
<td>obliged</td>
<td>$m_1$</td>
<td>1 minute</td>
</tr>
<tr>
<td>left</td>
<td>obliged</td>
<td>$m_2$</td>
<td>3 minute</td>
</tr>
<tr>
<td>right</td>
<td>obliged</td>
<td>$m_2$</td>
<td>1 day</td>
</tr>
<tr>
<td>attacker</td>
<td>obliged</td>
<td>$m_3$</td>
<td>30 seconds</td>
</tr>
</tbody>
</table>
1. Organisation is created (by the agents)
   - instances of groups
   - instances of schemes
2. Agents enter into groups adopting roles
3. Groups become responsible for schemes
   - Agents from the group are then obliged to commit to missions in the scheme
4. Agents commit to missions
5. Agents fulfil mission’s goals
6. Agents leave schemes and groups
7. Schemes and groups instances are destroyed
Organisation management infrastructure (OMI)

Responsibility

- Managing – coordination, regulation – the agents’ execution within organisation defined in an organisational specification

(e.g. MadKit, AMELI, S-Moise\(^+\), THOMAS, ...)

(e.g. MadKit, AMELI, S-Moise\(^+\), THOMAS, ...)

(e.g. MadKit, AMELI, S-Moise\(^+\), THOMAS, ...)

(e.g. MadKit, AMELI, S-Moise\(^+\), THOMAS, ...)

(e.g. MadKit, AMELI, S-Moise\(^+\), THOMAS, ...)

(e.g. MadKit, AMELI, S-Moise\(^+\), THOMAS, ...)

(e.g. MadKit, AMELI, S-Moise\(^+\), THOMAS, ...)

(e.g. MadKit, AMELI, S-Moise\(^+\), THOMAS, ...)

(e.g. MadKit, AMELI, S-Moise\(^+\), THOMAS, ...)

(e.g. MadKit, AMELI, S-Moise\(^+\), THOMAS, ...)

(e.g. MadKit, AMELI, S-Moise\(^+\), THOMAS, ...)

(e.g. MadKit, AMELI, S-Moise\(^+\), THOMAS, ...)

(e.g. MadKit, AMELI, S-Moise\(^+\), THOMAS, ...)

(e.g. MadKit, AMELI, S-Moise\(^+\), THOMAS, ...)

(e.g. MadKit, AMELI, S-Moise\(^+\), THOMAS, ...)

(e.g. MadKit, AMELI, S-Moise\(^+\), THOMAS, ...)

(e.g. MadKit, AMELI, S-Moise\(^+\), THOMAS, ...)

(e.g. MadKit, AMELI, S-Moise\(^+\), THOMAS, ...)

(e.g. MadKit, AMELI, S-Moise\(^+\), THOMAS, ...)

(e.g. MadKit, AMELI, S-Moise\(^+\), THOMAS, ...)

(e.g. MadKit, AMELI, S-Moise\(^+\), THOMAS, ...)

(e.g. MadKit, AMELI, S-Moise\(^+\), THOMAS, ...)

(e.g. MadKit, AMELI, S-Moise\(^+\), THOMAS, ...)

(e.g. MadKit, AMELI, S-Moise\(^+\), THOMAS, ...)

(e.g. MadKit, AMELI, S-Moise\(^+\), THOMAS, ...)

(e.g. MadKit, AMELI, S-Moise\(^+\), THOMAS, ...)

(e.g. MadKit, AMELI, S-Moise\(^+\), THOMAS, ...)

(e.g. MadKit, AMELI, S-Moise\(^+\), THOMAS, ...)

(e.g. MadKit, AMELI, S-Moise\(^+\), THOMAS, ...)

(e.g. MadKit, AMELI, S-Moise\(^+\), THOMAS, ...)

(e.g. MadKit, AMELI, S-Moise\(^+\), THOMAS, ...)

(e.g. MadKit, AMELI, S-Moise\(^+\), THOMAS, ...)

(e.g. MadKit, AMELI, S-Moise\(^+\), THOMAS, ...)

(e.g. MadKit, AMELI, S-Moise\(^+\), THOMAS, ...)

(e.g. MadKit, AMELI, S-Moise\(^+\), THOMAS, ...)

(e.g. MadKit, AMELI, S-Moise\(^+\), THOMAS, ...)

(e.g. MadKit, AMELI, S-Moise\(^+\), THOMAS, ...)

(e.g. MadKit, AMELI, S-Moise\(^+\), THOMAS, ...)

(e.g. MadKit, AMELI, S-Moise\(^+\), THOMAS, ...)

(e.g. MadKit, AMELI, S-Moise\(^+\), THOMAS, ...)

(e.g. MadKit, AMELI, S-Moise\(^+\), THOMAS, ...)

(e.g. MadKit, AMELI, S-Moise\(^+\), THOMAS, ...)

(e.g. MadKit, AMELI, S-Moise\(^+\), THOMAS, ...)

(e.g. MadKit, AMELI, S-Moise\(^+\), THOMAS, ...)

(e.g. MadKit, AMELI, S-Moise\(^+\), THOMAS, ...)

(e.g. MadKit, AMELI, S-Moise\(^+\), THOMAS, ...)

(e.g. MadKit, AMELI, S-Moise\(^+\), THOMAS, ...)

(e.g. MadKit, AMELI, S-Moise\(^+\), THOMAS, ...)

(e.g. MadKit, AMELI, S-Moise\(^+\), THOMAS, ...)

(e.g. MadKit, AMELI, S-Moise\(^+\), THOMAS, ...)

(e.g. MadKit, AMELI, S-Moise\(^+\), THOMAS, ...)

(e.g. MadKit, AMELI, S-Moise\(^+\), THOMAS, ...)

(e.g. MadKit, AMELI, S-Moise\(^+\), THOMAS, ...)

(e.g. MadKit, AMELI, S-Moise\(^+\), THOMAS, ...)

(e.g. MadKit, AMELI, S-Moise\(^+\), THOMAS, ...)

(e.g. MadKit, AMELI, S-Moise\(^+\), THOMAS, ...)

(e.g. MadKit, AMELI, S-Moise\(^+\), THOMAS, ...)

(e.g. MadKit, AMELI, S-Moise\(^+\), THOMAS, ...)

(e.g. MadKit, AMELI, S-Moise\(^+\), THOMAS, ...)

(e.g. MadKit, AMELI, S-Moise\(^+\), THOMAS, ...)

(e.g. MadKit, AMELI, S-Moise\(^+\), THOMAS, ...)

(e.g. MadKit, AMELI, S-Moise\(^+\), THOMAS, ...)

(e.g. MadKit, AMELI, S-Mois...
Organisational artifacts in ORA4MAS

- Based on A&A and Moise
- Agents create and handle organisational artifacts
- Artifacts in charge of regimentations, detection and evaluation of norms compliance
- Agents are in charge of decisions about sanctions
- Distributed solution
ORA4MAS – GroupBoard artifact

Observable Properties:

▶ **specification**: the specification of the group in the OS (an object of class moise.os.ss.Group)

▶ **players**: a list of agents playing roles in the group. Each element of the list is a pair (agent x role)

▶ **schemes**: a list of scheme identifiers that the group is responsible for
**Operations:**

- **adoptRole(role):** the agent executing this operation tries to adopt a role in the group
- **leaveRole(role)**
- **addScheme(schid):** the group starts to be responsible for the scheme managed by the SchemeBoard `schId`
- **removeScheme(schid)**
### Observable Properties:

- **specification**: the specification of the scheme in the OS
- **groups**: a list of groups responsible for the scheme
- **players**: a list of agents committed to the scheme. Each element of the list is a pair (agent, mission)
- **goals**: a list with the current state of the goals
- **obligations**: list of obligations currently active in the scheme
ORA4MAS – SchemeBoard artifact

Operations:

▸ commitMission(mission) and leaveMission: operations to “enter” and “leave” the scheme

▸ goalAchieved(goal): defines that some goal is achieved by the agent performing the operation

▸ setGoalArgument(goal, argument, value): defines the value of some goal’s argument
Agent integration

- Agents can interact with organisational artifacts as with ordinary artifacts by perception and action
- Any Agent Programming Language integrated with CArtAgO can use organisational artifacts

Agent integration provides some “internal” tools for the agents to simplify their interaction with the organisation:

- maintenance of a local copy of the organisational state
- production of organisational events
- provision of organisational actions
Organisational **actions** in *Jason*

Example (GroupBoard)

```plaintext
... joinWorkspace("ora4mas", O4MWsp);
makeArtifact(
    "auction",
    "ora4mas.nopl.GroupBoard",
    ["auction-os.xml", auctionGroup],
    GrArtId);
adoptRole(auctioneer);
focus(GrArtId);
...
```
Organisational **actions** in *Jason II*

Example (SchemeBoard)

```plaintext
...
makeArtifact(
    "sch1",
    "ora4mas.nopl.SchemeBoard",
    ["auction-os.xml", doAuction],
    SchArtId);
focus(SchArtId);
addScheme(Sch);
commitMission(mAuctioneer)[artifact_id(SchArtId)];
...
```
Organisational perception

When an agent focuses on an Organisational Artifact, the observable properties (Java objects) are translated to beliefs with the following predicates:

- specification
- play(agent, role, group)
- commitment(agent, mission, scheme)
- goalState(scheme, goal, list of committed agents, list of agents that achieved the goal, state of the goal)
- obligation(agent, norm, goal, deadline)
- ....
Organisational perception – example

**Inspection of agent bob (cycle #0)**

- Beliefs

  - commitment(bob,mManager,"sch2")
  - commitment(bob,mManager,"sch1")
  - current_wsp(cobj_1,"ora4mas","308b05b0-2994-4fe8-0331-0f1fcbbe2ebd")
  - formationStatus(ok)
  - goalState("sch2",wp,[bob],[bob],satisfied)
Handling organisational events in Jason

Whenever something changes in the organisation, the agent architecture updates the agent belief base accordingly producing events (belief update from perception)

Example (new agent entered the group)

```
+play(Ag,boss,GId) <- .send(Ag,tell,hello).
```

Example (change in goal state and norm violation)

```
+goalState(Scheme,wsecs,_,_,satisfied)
  : .my_name(Me) & commitment(Me,mCol,Scheme)
  <- leaveMission(mColaborator,Scheme).

+normFailure(N) <- .print("norm failure event: ", N).
```
Typical plans for obligations

+obligation(Ag,Norm,committed(Ag,Mission,Scheme),DeadLine)
  : .my_name(Ag)
  <- .print("I am obliged to commit to ",Mission);
  commitMission(Mission,Scheme).

+obligation(Ag,Norm,achieved(Sch,Goal,Ag),DeadLine)
  : .my_name(Ag)
  <- .print("I am obliged to achieve goal ",Goal);
  !Goal[scheme(Sch)];
  goalAchieved(Goal,Sch).

+obligation(Ag,Norm,What,DeadLine)
  : .my_name(Ag)
  <- .print("I am obliged to ",What,
             ", but I don't know what to do!").
Summary

- Ensures that the agents follow some of the constraints specified for the organisation
- Helps the agents to work together
- The organisation is **interpreted at runtime**, it is not hardwired in the agents code
- The agents ‘handle’ the organisation (i.e. their artifacts)
- It is suitable for open systems as no specific agent architecture is required

- available as open source at

  http://moise.sourceforge.net
Conclusions
Multiagent Systems

- MAS is an *organisation* of autonomous *agents* interacting together to achieve their goals within a shared *environment*.

- MAOP is a conceptual and practical tool to design and implement distributed, complex, huge, open, .... systems.
Agents

Programming **actions** with

- high level abstraction
  (beliefs, plans, goals, ...)
- concurrent, distributed, decoupled, open, ...
Programming **tools** for the agents

- high level abstraction
  (workspaces, artifacts, perception, action, ...)
- concurrent, distributed, decoupled, open, ...
Organisation

Helping the agents to live together

- high level abstraction
  (group, roles, schemes, norms, ...)
- concurrent, distributed, decoupled, open, ...
Main Abstractions

Organization
- Group
- Scheme
  - Role
  - Norm
  - Goal

Dimension
- Concept
  - dynamic relation
    - composition

Environment
- Workspace
- Signal
  - Artifact
  - Operation
  - Property

Agent
- Goal
  - Action
  - Belief

Environment
- Articulate
- Count-as
- Participate
- Coordinate
- Regulate
- Act
- Perceive
- Communicate
What we have learnt?

- MAS is not only agents
- MAS is not only organisation
- MAS is not only environment
- MAS is not only interaction

⇒ separation of concerns
⇒ the right tool for each problem
Acknowledgements

- Various colleagues and students
- JaCaMo users for helpful feedback
- CNPq, CAPES, ANP for supporting some of our current research


Bibliography V

The MADKIT agent platform architecture.

Moise: An organizational model for multi-agent systems.

Programming rational agents in GOAL.
In [Bordini et al., 2009], pages 119–157.

Formal semantics for an abstract agent programming language.

Instrumenting Multi-Agent Organisations with Organisational Artifacts and Agents.
Journal of Autonomous Agents and Multi-Agent Systems.


La méthode (1) : la nature de la nature.
Points Seuil.

Embodying organisations in multi-agent work environments.
In IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT 2009), Milan, Italy.

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

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

Environment programming in multi-agent systems – an artifact-based perspective.
Autonomous Agents and Multi-Agent Systems.
Published Online with ISSN 1573-7454 (will appear with ISSN 1387-2532).


A minimal dynamical organization model.

Artificial Intelligence, A Modern Approach (2nd ed.).
Prentice Hall.

Agent-oriented programming.

MASQ: towards an integral approach to interaction.
In AAMAS (2), pages 813–820.

Towards flexible teamwork.
Environment as a first-class abstraction in multi-agent systems.

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

*An Introduction to Multi-Agent Systems.*
John Wiley & Sons, Ltd.

*An Introduction to MultiAgent Systems.*
Agent Oriented Programming
- Smart Room
- Fundamentals
- Introduction to Jason
- Main constructs: beliefs, goals, and plans
- Reasoning Cycle
- Other language features
- Comparison with other paradigms

Environment Oriented Programming
- Fundamentals
- Existing approaches
  - Basic Level
  - Advanced Level
- Artifacts and CArtAgO
- CArtAgO and Agents (E-A)
- Conclusions and wrap-up
Organisation Oriented Programming
   Fundamentals
   Some OOP approaches
   The Moise framework
   Moise Organisation Modelling Language (OML)
   ORA4MAS Organisation Management Infrastructure (OMI)
   Jason and ORA4MAS integration

Multiagent Oriented Programming