Multi-Agent Oriented Programming using JaCaMo

Jomi F. Hübner

Federal University of Santa Catarina, Brazil

Summer School on AI for Industry 4.0
Saint-Etienne 2020
Context

From knowledge to **action**

From theoretical to **practical** reasoning

From mind to body & environment & others (interaction)

From individuals to **societies**
Context

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
  ▶ Jason
▶ Environment
▶ Organisation
▶ MAOP

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

Books:  [Bordini et al., 2005], [Bordini et al., 2009]
Proceedings:  ProMAS, DALT, LADS, EMAS, AGERE, ...
Surveys:  [Bordini et al., 2006], [Fisher et al., 2007] ...
Languages of historical importance:  Agent0 [Shoham, 1993],
AgentSpeak(L) [Rao, 1996], MetateM [Fisher, 2005],
3APL [Hindriks et al., 1997],
Golog [Giacomo et al., 2000]
Other prominent languages:
  Jason [Bordini et al., 2007],
  Jadex [Pokahr et al., 2005], 2APL [Dastani, 2008],
  GOAL [Hindriks, 2009], JACK [Winikoff, 2005],
  JIAC, ASTRA
But many others languages and platforms...
Some Languages and Platforms

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

Features

- Reacting to events \times long-term goals
- Course of actions depends on circumstance
- Plan failure (dynamic environments)
- Social ability
- Combination of theoretical and practical reasoning
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]
Motivation for BDI — autonomous robot

[Coen and Levesque, 1990]

A household robot.¹ You say “Willie, bring me a beer.” The robot replies “OK, boss.” Twenty minutes later, you screech “Willie, why didn’t you bring that beer?” It answers “Well, I intended to get you the beer, but I decided to do something else.” Miffed, you send the wise guy back to the manufacturer, complaining about a lack of commitment. After retrofitting, Willie is returned, marked “Model C: The Committed Assistant.” Again, you ask Willie to bring a beer. Again, it accedes, replying “Sure thing.” Then you ask: “What kind did you buy?” It answers: “Genesee.” You say “Never mind.” One minute later, Willie trundles over with a Genesee in its gripper. This time, you angrily return Willie for overcommitment. After still more tinkering, the manufacturer sends Willie back, promising no more problems with its commitments. So, being a somewhat trusting consumer, you accept the rascal back into your household, but as a test, you ask it to bring you your last beer. Willie again accedes, saying “Yes, Sir.” (Its attitude problem seems to have been fixed.) The robot gets the beer and starts towards you. As it approaches, it lifts its arm, wheels around, deliberately smashes the bottle, and trundles off. Back at the plant, when interrogated by customer service as to why it had abandoned its commitments, the robot replies that according to its specifications, it kept its commitments as long as required—commitments must be dropped when fulfilled or impossible to achieve. By smashing the last bottle, the commitment
BDI architecture
(the mentalistic view)
while true do
    $B \leftarrow brf(B, \text{perception}())$ // belief revision
    $D \leftarrow \text{options}(B, I)$ // desire revision
    $I \leftarrow \text{deliberate}(B, D, I)$ // get intentions
    $\pi \leftarrow \text{meansend}(B, I, A)$ // gets a plan
while $\pi \neq \emptyset$ do
    execute( head($\pi$) )
    $\pi \leftarrow \text{tail}(\pi)$
while true do

\[ B \leftarrow \text{brf}(B, \text{perception}()) \]  \hspace{2cm} // belief revision

\[ D \leftarrow \text{options}(B, I) \]  \hspace{2cm} // desire revision

\[ I \leftarrow \text{deliberate}(B, D, I) \]  \hspace{2cm} // get intentions

\[ \pi \leftarrow \text{meansend}(B, I, A) \]  \hspace{2cm} // gets a plan

while \( \pi \neq \emptyset \) do

execute(\( \text{head}(\pi) \))

\[ \pi \leftarrow \text{tail}(\pi) \]

fine for pro-activity, but not for reactivity (over commitment)
while true do
  B ← brf(B, perception())  // belief revision
  D ← options(B, I)          // desire revision
  I ← deliberate(B, D, I)    // get intentions
  π ← meansend(B, I, A)      // gets a plan
  while π ≠ ∅ do
    execute( head(π) )
    π ← tail(π)
    B ← brf(B, perception())
    if ¬sound(π, I, B) then
      π ← meansend(B, I, A)
  end while
  B ← brf(B, perception())

revise commitment to plan – re-planning for context adaptation
while true do

1. $B \leftarrow \text{brf}(B, \text{perception}())$ // belief revision
2. $D \leftarrow \text{options}(B, I)$ // desire revision
3. $I \leftarrow \text{deliberate}(B, D, I)$ // get intentions
4. $\pi \leftarrow \text{meansend}(B, I, A)$ // gets a plan
5. while $\pi \neq \emptyset$ and $\neg\text{succeeded}(I, B)$ and $\neg\text{impossible}(I, B)$ do
6.   execute(head($\pi$))
7.   $\pi \leftarrow \text{tail}(\pi)$
8.   $B \leftarrow \text{brf}(B, \text{perception}())$
9.   if $\neg\text{sound}(\pi, I, B)$ then
10.      $\pi \leftarrow \text{meansend}(B, I, A)$
11. revise commitment to intentions – Single-Minded Commitment
while true do

\[ B \leftarrow \text{brf}(B, \text{perception}()) \] // belief revision

\[ D \leftarrow \text{options}(B, l) \] // desire revision

\[ I \leftarrow \text{deliberate}(B, D, l) \] // get intentions

\[ \pi \leftarrow \text{meansend}(B, I, A) \] // gets a plan

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 reconsider(\( I, B \)) then

\[ D \leftarrow \text{options}(B, l) \]

\[ I \leftarrow \text{deliberation}(B, D, l) \]

if \( \neg \text{sound}(\pi, I, B) \) then

\[ \pi \leftarrow \text{meansend}(B, I, A) \]

reconsider the intentions (not always!)
Jason

(let’s go programming those nice concepts)
happy(bob).  // B
!say(hello).    // D

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

(BDI & Jason) Hello World – agent bob
(BDI & Jason) Hello World – agent bob

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

beliefs

▶ prolog like (FOL)

// B

// D

// I
happy(bob).
!say(hello).

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

- desires
  ▶ prolog like
  ▶ with ! prefix

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

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

plans

- define when a desire becomes an intention
  ⇝ deliberate
- how it is satisfied
- are used for practical reasoning
  ⇝ means-end
+happy(bob) <- !say(hello).

+!say(X) : not today(monday)
  <- .print(X).
source of beliefs

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

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

plan selection

\[ \text{+happy}(H) [\text{source}(A)] \]
\[ : \quad \text{sincere}(A) \land .\text{my\_name}(H) \]
\[ \leftarrow .\text{say}(\text{hello}). \]

\[ \text{+happy}(H) \]
\[ : \quad \text{not} \quad .\text{my\_name}(H) \]
\[ \leftarrow .\text{say}(\text{i\_envy}(H)). \]

\[ .\text{say}(X) : \quad \text{not} \quad \text{today}(\text{monday}) \leftarrow .\text{print}(X). \]
Hello World
intention revision

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

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

+!say(X) : not today(monday) <- .print(X); !say(X).
Hello World
intention revision

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

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

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

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

intention revision

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

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

+!say(X) : not today(monday)

-happy(H)
    : .my_name(H)
    <- .drop_intention(say(hello)).
```

features

- we can have several intentions based on the same plans
  ⇝ running concurrently
- long term goals running
  ⇝ reaction meanwhile
  ⇝ not overcommitted
- plan selection based on circumstance
- actions (partially) computed by the interpreter
  ⇝ programmer declares plans
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
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)

\[
\begin{align*}
\text{red(box1)} & [\text{source(percept)}]. \\
\text{friend(bob,alice)} & [\text{source(bob)}]. \\
\text{lier(alice)} & [\text{source(self)}, \text{source(bob)}]. \\
\sim \text{lier(bob)} & [\text{source(self)}].
\end{align*}
\]
Beliefs — Dynamics I

by perception

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

by intention

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

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

`-lier(john); // removes lier(john)[source(self)]`
when an agent receives a \textbf{tell} message, the content is a new belief annotated with the sender of the message

```
.send(tom,tell,lier(alice)); // sent by bob
// adds \textit{lier(alice)[source(bob)]} in Tom's BB
...
.send(tom,untell,lier(alice)); // sent by bob
// removes \textit{lier(alice)[source(bob)]} from Tom's BB
```
Goals — Representation

Types of goals

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

Syntax

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

Example (Initial goal of agent Tom)

!write(book).
Goals — Dynamics

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);
...```

by intention
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
```
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:

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

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)
Plans — Operators for Plan Body

\[
+\text{rain} : \ time\_to\_leave(T) \ & \ clock]\text{.}\text{now}(H) \ & \ H \geq T
\]
\[
\leftarrow \ !g1; \quad \text{// new sub-goal}
\]
\[
!!g2; \quad \text{// new goal}
\]
\[
?b(X); \quad \text{// new test goal}
\]
\[
+b1(T-H); \quad \text{// add mental note}
\]
\[
-b2(T-H); \quad \text{// remove mental note}
\]
\[
+-b3(T*H); \quad \text{// update mental note}
\]
\[
jia.\text{get}(X); \quad \text{// internal action}
\]
\[
X > 10; \quad \text{// constraint to carry on}
\]
\[
\text{close(door);} \quad \text{// external action}
\]
\[
!g3[\text{hard}\_\text{deadline}(3000)]. \quad \text{// goal with deadline}
\]
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`
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. perceive

2. BUF

3. checkMail

4. SocAcc

5. $S_k$

6. $S_j$

7. Check Context

8. Intended Means

9. Selected Intention

10. Execute Intention

- Beliefs to Add and Delete
- External Events
- Internal Events
- Relevant Plans
- Plans
- Applicable Plans
- Intended Means
- Selected Event
- Relevant Plans
- Selected Event
- New Plan
- Percepts
- Messages
- Suspended Intentions (Actions and Msgs)
- Intentions
- Push New Plan
- New Intentions
- Updated Intention
- Actions
- Messages
- send
- sendMsg
- Plan Library

Agent
Jason Reasoning Cycle

- Percepts
- Messages

1. perceive
2. BUF
3. checkMail
4. SocAcc

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

_f_ is the drop condition for goal _g_.
Compiler pre-processing – directives

Example (single minded commitment)

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

Example (an agent that asks for plans on demand)

```c
%!G[error(no_relevant)] : teacher(T)
  <- .send(T, askHow, { +!G }, Plans);
  .add_plan(Plans);
  !G.
```

_in the event of a failure to achieve any goal $G$ due to no relevant plan, asks a teacher for plans to achieve $G$ and then try $G$ again_

- The failure event is annotated with the error type, line, source, ... _error(no_relevant)_ means no plan in the agent’s plan library to achieve $G$
- `{ +!G }` is the syntax to enclose triggers/plans as terms
Other Language Features

Strong Negation

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

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

  `libname.action_name(...)"
Standard Internal Actions

- Standard (pre-defined) internal actions have an empty library name
  - `.print(term_1, term_2, ...)`
  - `.union(list_1, list_2, list_3)`
  - `.my_name(var)`
  - `.send(ag, perf, literal)`
  - `.intend(literal)`
  - `.drop_intention(literal)`

- Many others available for: printing, sorting, list/string operations, manipulating the beliefs/annotations/plan library, creating agents, waiting/generating events, etc.
Namespaces & Modularity
Namespaces & Modularity

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(self)].
propose(12.043311087442898)[source(self)].
propose(12.81277904935436)[source(self)].
winner(company_A1)[source(self)].

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

pc::
introduction(participant)[source(comparator)].
propose(11.389500048463455)[source(self)].
propose(11.392553683771682)[source(self)].
propose(12.34890100262853)[source(self)].
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, ...)

52
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();
            }
        }
    }
}
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(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
  - Open issues
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
Single Agent Perspective

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

AGENTs

MAS ENVIRONMENT

SIMULATED WORLD

or

REAL WORLD

(PHYSICAL OR COMPUTATIONAL)

or

EXTERNAL WORLD

(PHYSICAL OR COMPUTATIONAL)

or

WRAPPER TO EXISTING TECHNOLOGY

Example:
JAVA PLATFORM AGENTS

actions

percepts

mimicking
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 CArtaGOno
Agents and Artifacts (A&A) Conceptual Model: Background Human Metaphor
A&A Meta-Model in More Detail [Ricci et al., 2010a]
Artifact Abstract Representation
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** ↔ **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** ↔ **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
Interaction Model: Observation

- 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
Wrap-up

- 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]
  $\Rightarrow$ 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]
  $\Rightarrow$ 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]
  $\Rightarrow$ 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]
  $\Rightarrow$ pattern of emergent cooperation
Organisation in MAS – a definition

- Pattern of agent *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
Organisation Oriented Programming (OOP)

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 Specification

Organisation Entity

Agent

Agent

Agent
Organisation Oriented Programming (OOP)

Components
- Programming language (OML)
- Platform (OMI)
- Integration to agent architectures and to environment
Motivations for OOP: Applications point of view

- Current applications show an increase in:
  - Number of agents
  - Duration and repetitiveness of agent activities
  - Heterogeneity of the agents
  - Number of designers of agents
  - Agent ability to act and decide
  - Openness, scalability, dynamism

- More and more applications require the integration of human communities and technological communities (ubiquitous and pervasive computing), building connected communities (ICities) in which agents act on behalf of users:
  - Trust, security, ..., flexibility, adaptation
Motivations for OOP: 
**Normative** point of view

- MAS have two properties which seem contradictory:
  - a **global** purpose
  - **autonomous** agents
  
  While the autonomy of the agents is essential, it may cause loss in the global coherence of the system and achievement of the global purpose

- Embedding **norms** within the **organisation** of an MAS is a way to constrain the agents’ behaviour towards the global purposes of the organisation, while explicitly addressing the autonomy of the agents within the organisation
  
  Normative organisation
  
  e.g. when an agent adopts a role, it adopts a set of behavioural constraints that support the global purpose of the organisation.
  
  It may decide to obey or disobey these constraints
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 (\(\mathcal{J}\)-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
Graphical representation of structural specification of 3-5-2 Joj Team

Organizational Entity

- Dida: goalkeeper
- Lucio: back
- Juan: back
- Cafu: leader
- Kaka: middle
- Emerson: middle
- Ze Roberto: attacker
- Ronaldinho: attacker
- Roberto Carlos: attacker
- Ronaldo: attacker
- Adriano: attacker
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

get the ball

m1

go towards the opponent field

m1

be placed in the middle field

m2

be placed in the opponent goal area

m3

be placed in the middle field

m2

kick the ball to the goal area

m3

kick the ball to (agent committed to m2)

Key

Organizational Entity

Lucio  m1
Cafu  m2
Rivaldo  m3

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>$m_1$ get the ball, go ...</td>
<td>1 minute</td>
</tr>
<tr>
<td>left</td>
<td>obliged</td>
<td>$m_2$ be placed at ..., kick ...</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>$m_3$ kick to the goal, ...</td>
<td>30 seconds</td>
</tr>
</tbody>
</table>
Organisational Entity

- Structural groups, links, roles
- Normative norms
- Functional schemas, missions
- Purpose

- Group instances
- Role player
- Schema instances
- Mission player
- Agents

Organisation specification

Organisation Entity
Organisation Entity Dynamics

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, ...)

Diagram:
- Organisation Program
- OMI
- Agent (Multiple)

The diagram shows the interaction between the Organisation Program, OMI, and multiple agents. The Organisation Program feeds into the OMI, which then manages the agents. The agents are connected to each other and to the OMI, indicating a networked system for coordination and regulation.
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
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
ORA4MAS – GroupBoard artifact

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)`
ORA4MAS – SchemeBoard artifact

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)

...
joinWorkspace("ora4mas", O4MWsp);
makeArtifact(
  "auction",
  "ora4mas.nopl.GroupBoard",
  ["auction-os.xml", auctionGroup],
  GrArtId);
adoptRole(auctioneer);
focus(GrArtId);
...

Organisational **actions** in *Jason II*

### Example (SchemeBoard)

...  
```javascript
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") [artifact_id(cobj_4),concept], artifact_name(cobj_4,"sch2"), artifact_type(cobj_4,"ora4m commitment(bob,mManager,"sch1") [artifact_id(cobj_3),concept], artifact_name(cobj_3,"sch1"), artifact_type(cobj_3,"ora4m current_wsp(cobj_1,"ora4mas","308b05b0-2994-4fe8 formationStatus(ok) [artifact_id(cobj_2),obs_prop_id("obs_i, obj_2,"mypaper"), artifact_type(cobj_2,"ora4mas.nopl.GroupBo goalState("sch2",wp,[bob],[bob],satisfied) [artifact_id(cot
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)**

\[+\text{play}(\text{Ag}, \text{boss}, \text{GId}) \leftarrow .\text{send}(\text{Ag}, \text{tell}, \text{hello}).\]

**Example (change in goal state and norm violation)**

\[+\text{goalState}(\text{Scheme}, \text{wsecs}, \_, \_, \_, \text{satisfied})
  : .\text{my
ame}(\text{Me}) \& \text{commitment}(\text{Me}, \text{mCol}, \text{Scheme})
  \leftarrow \text{leaveMission}(\text{mColaborator}, \text{Scheme}).\]

\[+\text{normFailure}(\text{N}) \leftarrow .\text{print}("\text{norm failure event: }", \text{N}).\]
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 – Moise

- 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

- All 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
org
mission
schema
ORGAMISATION
LEVEL
AGENT
LEVEL
ENDOGENOUS
ENVIRONMENT
LEVEL
wsp
artifact
network node
EXOGENOUS
ENVIRONMENT
agent
agent
artifact
wsp
network node
EXOGENOUS
ENVIRONMENT
AGENT
LEVEL
ENDOGENOUS
ENVIRONMENT
LEVEL
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, ...
Helping the agents to live **together**

- high level abstraction
  (group, roles, schemes, norms, ...)
- concurrent, distributed, decoupled, open, ...
What we have learnt in this project?

- 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
Further Resources

▶ http://jacamo.sourceforge.net

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

- Various colleagues and students
- JaCaMo users for helpful feedback
- CNPq, CAPES, ANP for supporting some of our current research
La sociologie des organisations. 
Seuil, 3ème edition.

A survey of programming languages and platforms for multi-agent systems. 
Informatica (Slovenia), 30(1):33–44.

Multi-Agent Programming: Languages, Platforms and Applications, volume 15 of 
Multiagent Systems, Artificial Societies, and Simulated Organizations. 
Springer.

Multi-Agent Programming: Languages, Tools and Applications. 
Springer.

Programming Multi-Agent Systems in AgentSpeak Using Jason. 


**Metatem: The story so far.**  

**Computational logics and agents: A road map of current technologies and future trends.**  

**Organizations in multi-agent systems.**  
In *Pre-Proceeding of the 10th European Worshop on Modeling Autonomous Agents in a Multi-Agent World (MAAMAW’2001)*, Annecy.

**Moiseinst: An organizational model for specifying rights and duties of autonomous agents.**  

**Congolog, a concurrent programming language based on the situation calculus.**  


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.
*Journal of Artificial Intelligence Reseearch, 7*:83–124.


Agent Oriented Programming

- Fundamentals
- (BDI) Hello World
- 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