Olivier Boissier\textsuperscript{1}, Rafael H. Bordini\textsuperscript{2},
Jomi F. Hübner\textsuperscript{3}, Alessandro Ricci\textsuperscript{4}

\textsuperscript{1}EMSE, France
Olivier.Boissier@emse.fr

\textsuperscript{2}PUC-RS, Brazil
R.Bordini@pucrs.br

\textsuperscript{3}DAS-UFSC, Brazil
jomi@das.ufsc.br

\textsuperscript{4}University of Bologna, Italy
a.ricci@unibo.it
Outline of the seminar

- Introduction
- AOP
  - About Agent Oriented Programming
  - *Jason*
- EOP
  - About Environment Oriented Programming
  - A&A and CArtAgO
- OOP
  - About Organisation Oriented Programming
  - *Moise*
- Putting the pieces together: *JaCaMo*
Introduction
Abstractions in Multi-Agent Systems
Abstractions in Multi-Agent Systems

- **Individual Agent Level**: autonomy, situatedness
  - **Cognitive Concepts**: beliefs, desires, goals, intentions, plans
  - **Reasoning Cycle**: sense/reason/act, reactive/pro-active behaviour

- **Environment Level**: resources and services that agents can access and control; sensing and acting in an environment

- **Social and Organisation Level**: cooperation, coordination, regulation patterns
  - **Roles**: rights, responsibilities, ...
  - **Organisational Rules**: constraints on roles and their interactions, norms, deadlines, ...
  - **Organisational Structures**: topology of interaction patterns and relations over activity control
Proposed by Shoham [Shoham, 1993]
Use of mentalistic notions and a societal view of computation (anthropomorphism)
Levels of abstraction: Agents – Organisations – Environment
Programming languages for agents have developed a lot since then, but still not a mature paradigm
Programming languages/platforms for organisation and environment are also being developed
Some agent development platforms have a formal basis
Many influenced by the BDI agent architecture
BDI Architecture

- Intentional Stance (Dennett)
- Practical Reasoning (Bratman)
- IRMA (Bratman, Isreal, Pollack)
- PRS (Georgeff, Lansky)
- dMARS (Kinny)
- BDI Logics and Agent Architecture (Rao, Georgeff)
- Wooldridge, Singh, ...
Programming Languages for Multi-Agent Systems
E.g., Jason, Jadex, JACK, 2APL, GOAL, Brahms, JIAC, Agent Factory, MetateM, Golog variants, ...

- **Architecture** to represent an agent mental state:
  - **Beliefs**: information available to agent (e.g., about the environment or other agents)
  - **Goals**: states of affairs that the agent wants to achieve
  - **Events**: changes in agents beliefs or goals
  - **Capabilities**: reusable modules of activities that the agent can perform
  - **Plans**: reasoning about courses of action to achieve goals
  - **Rules**: reasoning about beliefs
Some steps of a **Reasoning Cycle**:

- Determining Relevant Plans for Handling Events
- Select a Plan for Execution
- Execute Part of an Intended Plans
- Handle Plan Failures

**Agent Interpreter** is an infinite loop of such reasoning cycles. The architecture and reasoning cycle together with the agent program (specially plans) determine the behaviour of the agent.
Programming Languages/Platforms for Organisations

- **Concepts** used to specify the state of an organisation:
  - Agents, Roles, Groups
  - Norms, Obligations, Prohibitions, Permissions, Violations
  - Dependency, Power, Delegation, Information flow relations
  - Deadlines, Sanctions, Rewards

- **Management Infrastructure** to control and coordinate agent behaviour at run-time:
  - Endogenous: The control is a part of the agent program
  - Exogenous: The control is performed by an external system
    - Monitoring Agent Behaviour
    - Enforcing Organisational Rules
    - Regimenting Organisational Rules
Programming Languages/Platforms for Environments

- **Artifacts** to represent the state of the environment
  - Access to Databases/Services/etc., Coordination, Interaction
  - Environment “objects”, i.e., non-proactive entities
- **Processing Operations** on Artifacts
  - Realising the effects of environments actions
  - Providing events related to sensing the environment
  - Synchronising agent actions
- At the right level of abstraction for a multi-agent system
Putting the Pieces Together: *JaCaMo*

- First fully operational, unified platform covering the 3 main levels of abstractions for multi-agent oriented programming
- JaCaMo = Jason + CArtAgO + Moise
- [http://jacamo.sourceforge.net](http://jacamo.sourceforge.net)
  - [http://jason.sourceforge.net](http://jason.sourceforge.net)
  - [http://cartago.sourceforge.net](http://cartago.sourceforge.net)
  - [http://moise.sourceforge.net](http://moise.sourceforge.net)
- More than the sum of 3 successful platforms
- Revealing the full potential of **Multi-Agent Oriented Programming**
AOP
Outline

2 AOP: Agent Oriented Programming
- About AOP
- *Jason*
  - Introduction to *Jason*
  - Reasoning Cycle
  - Main Language Constructs: Beliefs, Goals, and Plans
  - Other Language Features
  - Comparison With Other Paradigms
  - The *Jason* Platform
  - Perspectives: Some Past and Future Projects
- Conclusions
About AOP
Agent Oriented Programming

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

- Heavily influence by the BDI architecture and reactive planning systems

- Various language constructs for the sophisticated abstractions used in AOSE
  - Agent: Belief, Goal, Intention, Plan
  - Organisation: Group, Role, Norm, Interactions
  - Environment: Artifacts, Percepts, Actions
Agent Oriented Programming

Features

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

Books: [Bordini et al., 2005a], [Bordini et al., 2009]

Proceedings: ProMAS, DALT, LADS, ... [Baldoni et al., 2010, Dastani et al., 2010, Hindriks et al., 2009, Baldoni et al., 2009, Dastani et al., 2008b, Baldoni et al., 2008, Dastani et al., 2008a, Bordini et al., 2007b, Baldoni and Endriss, 2006, Bordini et al., 2006b, Baldoni et al., 2006, Bordini et al., 2005b, Leite et al., 2005, Dastani et al., 2004, Leite et al., 2004]

Surveys: [Bordini et al., 2006a], [Fisher et al., 2007] ...

Languages of historical importance: Agent0 [Shoham, 1993], AgentSpeak(L) [Rao, 1996], MetateM [Fisher, 2005], 3APL [Hindriks et al., 1997], Golog [Giacomo et al., 2000]

Other prominent languages: Jason [Bordini et al., 2007c], Jadex [Pokahr et al., 2005], 2APL [Dastani, 2008a], GOAL [Hindriks, 2009], JACK [Winikoff, 2005]

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); ...
Already the **right** way to implement MAS is to use an AOSE Methodology (Prometheus, Gaia, Tropos, ...) and an MAS Programming Language!

Many agent languages have efficient and stable interpreters — used extensively in teaching

All have some programming tools (IDE, tracing of agents’ mental attitudes, tracing of messages exchanged, etc.)

Finally integrating with **social** aspects of MAS

Growing user base
Jason
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. Hbner** and **Rafael H. Bordini**
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
Main Architectural Components

Belief base: where beliefs are stored

Set of events: to keep track of events the agent will have to handle

Plan library: stores all the plans currently known by the agent

Set of Intentions: each intention keeps track of the goals the agent is committed to and the courses of action it chose in order to achieve the goals for one of various foci of attention the agent might have
Jason Interpreter
Basic Reasoning cycle

- perceive the environment and update belief base
- process new messages
- select event
- select **relevant** plans
- select **applicable** plans
- create/update intention
- select intention to execute
Jason Rereasoning Cycle

1. **perceive**
   - Percepts
   - BUF
   - BRF
   - External Events
   - Beliefs to Add and Delete

2. **SocAcc**
   - Events

3. **checkMail**
   - Percepts
   - Messages
   - S_M

4. **Belief Base**
   - Beliefs
   - External Events
   - Internal Events

5. **Selected Event**
   - Events
   - Selected Event
   - Relevant Plans
   - Plans

6. **Unify Event**
   - Beliefs
   - Intended Means
   - Intended Plans

7. **Check Context**
   - Beliefs
   - Applicable Plans
   - New Plans

8. **New**
   - Intention
   - Push New Plan
   - Suspended Intentions
   - (Actions and Msgs)

9. **Selected Intention**
   - Intentions
   - New Intention
   - New

10. **Execute Intention**
    - Action
    - Act
    - Messages
    - Suspended Intentions
    - (Actions and Msgs)

Agent

Plan Library

Message Library

Actions

...
Beliefs — Representation

Syntax

Beliefs are represented by annotated literals of first order logic

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

Example (belief base of agent Tom)

red(box1)[source(percept)].
friend(bob,alice)[source(bob)].
lier(alice)[source(self),source(bob)].
\neg lier(bob)[source(self)].
Beliefs — Dynamics I

**by perception**

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

**by intention**

the **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

```
.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).
Goals — Dynamics I

by intention

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

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

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

by communication — achievement goal

when an agent receives an **achieve** message, the content is a new achievement goal annotated with the sender of the message

```
.send(tom,achieve,write(book)); // sent by Bob
// adds new goal write(book)[source(bob)] for Tom
...
.send(tom,unachieve,write(book)); // sent by Bob
// removes goal write(book)[source(bob)] for Tom
```
Goals — Dynamics III

by communication – test goal

when an agent receives an askOne or askAll message, the content is a new test goal annotated with the sender of the message

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

\[ \text{triggering_event} : \text{context} \gets \text{body}. \]

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

<table>
<thead>
<tr>
<th>Boolean operators</th>
<th>Arithmetic operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp; (and)</td>
<td>+ (sum)</td>
</tr>
<tr>
<td></td>
<td>(or)</td>
</tr>
<tr>
<td>not (not)</td>
<td>* (multiply)</td>
</tr>
<tr>
<td>= (unification)</td>
<td>/ (divide)</td>
</tr>
<tr>
<td>&gt;, &gt;= (relational)</td>
<td>div (divide – integer)</td>
</tr>
<tr>
<td>&lt;, &lt;= (relational)</td>
<td>mod (remainder)</td>
</tr>
<tr>
<td>== (equals)</td>
<td>** (power)</td>
</tr>
<tr>
<td>== (different)</td>
<td></td>
</tr>
</tbody>
</table>
A plan body may contain:

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

Example (plan body)

```
+rain : time_to_leave(T) & clock.now(H) & H >= T
  <- !g1;       // new sub-goal
  !!g2;         // new goal
  ?b(X);        // new test goal
  +b1(T-H);     // add mental note
  -b2(T-H);     // remove mental note
  -+b3(T*H);    // update mental note

/> jia.get(X); // internal action
X > 10;       // constraint to carry on
close(door).  // external action
```
Plans — Example

\begin{itemize}
  \item \texttt{+green_patch(Rock)[source(percept)]}
    \begin{itemize}
      \item \texttt{not battery\_charge(low)}
      \item \texttt{<- ?location(Rock,Coordinates);}
        \begin{itemize}
          \item \texttt{!at(Coordinates)};
          \item \texttt{!examine(Rock)}.
        \end{itemize}
    \end{itemize}

  \item \texttt{+!at(Coords)}
    \begin{itemize}
      \item \texttt{not at(Coords) \& safe\_path(Coords)}
      \item \texttt{<- move\_towards(Coords)};
        \begin{itemize}
          \item \texttt{!at(Coords)}.
        \end{itemize}
    \end{itemize}

  \item \texttt{+!at(Coords)}
    \begin{itemize}
      \item \texttt{not at(Coords) \& not safe\_path(Coords)}
      \item \texttt{<- ...}
    \end{itemize}

  \item \texttt{+!at(Coords) : at(Coords)}.
\end{itemize}
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
Strong Negation

Example

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

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

\[
\text{likely\_color(Obj,C) :-}
\text{  colour(Obj,C)[degOfCert(D1)] \&}
\text{  not (colour(Obj,\_)[degOfCert(D2)] \& D2 > D1) \&}
\text{  not \sim\text{colour(C,B)}.}
\]
Plan Annotations

- Like beliefs, plans can also have annotations, which go in the plan label.
- Annotations contain meta-level information for the plan, which selection functions can take into consideration.
- The annotations in an intended plan instance can be changed dynamically (e.g. to change intention priorities).
- There are some pre-defined plan annotations, e.g. to force a breakpoint at that plan or to make the whole plan execute atomically.

Example (an annotated plan)

@myPlan[chance_of_success(0.3), usual_payoff(0.9), any_other_property]
+!g(X) : c(t) <- a(X).
Example (an agent blindly committed to \( g \))

\[+!g : g.\]

\[+!g : \ldots \leftarrow \ldots ?g.\]

\[-!g : \text{true} \leftarrow !g.\]
Meta Programming

Example (an agent that asks for plans on demand)

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

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

- The failure event is annotated with the error type, line, source, ... *error(no\_relevant)* means no plan in the agent’s plan library to achieve G
- \{ +!G \} is the syntax to enclose triggers/plans as terms
Internal Actions

- Unlike actions, internal actions do not change the environment
- Code to be executed as part of the agent reasoning cycle
- AgentSpeak is meant as a high-level language for the agent’s practical reasoning and internal actions can be used for invoking legacy code elegantly

- Internal actions can be defined by the user in Java

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

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

- Many others available for: printing, sorting, list/string operations, manipulating the beliefs/annotations/plan library, creating agents, waiting/generating events, etc.
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
Example (Java code – go to gold)

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

(how to code the charge battery behaviour?)
Example (Java code – charge battery)

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

(note where the tests for low battery have to be done)
Example (*Jason* code)

```plaintext
+see(gold)
  <- !goto(gold).

+!goto(gold) : see(gold)       // long term goal
  <- !select_direction(A);
    go(A);
    !goto(gold).

+battery(low)                   // reactivity
  <- !charge.

^!charge[state(started)]       // goal meta-events
  <- .suspend(goto(gold)).

^!charge[state(finished)]
  <- .resume(goto(gold)).
```
With the *Jason* extensions, nice separation of theoretical and practical reasoning

BDI architecture allows
- long-term goals (goal-based behaviour)
- reacting to changes in a dynamic environment
- handling multiple foci of attention (concurrency)

Acting on an environment and a higher-level conception of a distributed system
Various communication and execution management infrastructures can be used with *Jason*:

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

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

**Jade**: distributed agents, FIPA-ACL

**Saci**: distributed agents, KQML

... others defined by the user (e.g. AgentScape)
Definition of a Simulated Environment

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

Environment Simulator

Agent Architecture

Reasoner

getPercepts

perceive

executeAction

act

change percept
Example of an Environment Class

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

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

```
Simple way of defining a multi-agent system

Example (MAS that uses JADE as infrastructure)

```java
MAS my_system {
  infrastructure: Jade
  environment: robotEnv
  agents:
    c3po;
    r2d2 at jason.sourceforge.net;
    bob #10; // 10 instances of bob
  classpath: "../lib/graph.jar";
}
```
Configuration of event handling, frequency of perception, user-defined settings, customisations, etc.

Example (MAS with customised agent)

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

```
MAS grid_world {

    environment: alice.c4jason.CEnv

    agents:
        cleanerAg
            agentArchClass alice.c4jason.CogAgentArch
            #3;
}
```
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, ...)
jEdit Plugin

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

$ +!get(beer)
$ +has(owner,beer)
$ +!check_bored.
$ +msg(M)[source]
$ -has(owner,beer)
$ +!drink(beer)

+!get(beer) : true
  $ -.send(robot, achieve, has(owner,beer)).
+has(owner,beer) : true
  $ -.drink(beer).
+!check_bored
  $ -.send(robot, bored, has(owner,beer)).
+has(owner,beer) : true
  $ -.get(beer).

// while I have beer, sip
+!drink(beer) : has(owner,beer)
```

Jason functions are also available in the Plugin-Jason menu.
Eclipse Plugin
Some Related Projects I

- **Speech-act** based communication  
  Joint work with Renata Vieira, Álvaro Moreira, and Mike Wooldridge

- **Cooperative** plan exchange  
  Joint work with Viviana Mascardi, Davide Ancona

- **Plan Patterns** for Declarative Goals  
  Joint work with M. Wooldridge

- **Planning** (Felipe Meneguzzi and Colleagues)

- **Web and Mobile Applications** (Alessandro Ricci and Colleagues)

- **Belief Revision**  
  Joint work with Natasha Alechina, Brian Logan, Mark Jago
Some Related Projects II

- **Ontological** Reasoning
  - Joint work with Renata Vieira, Álvaro Moreira
  - **JASDL**: joint work with Tom Klapiscak

- Goal-Plan Tree Problem (Thangarajah et al.)
  Joint work with Tricia Shaw

- Trust reasoning (ForTrust project)

- Agent verification and model checking
  Joint project with M.Fisher, M.Wooldridge, W.Visser, L.Dennis, B.Farwer
Environments, Organisation and Norms

- Normative environments
  - Join work with A.C.Rocha Costa and F.Okuyama
- MADeM integration (Francisco Grimaldo Moreno)
- Normative integration (Felipe Meneguzzi)
- \textbf{CArtAgO} integration
- \textbf{Moise}^+ integration

More on jason.sourceforge.net, related projects
Some Trends for *Jason* 1

- Modularity and encapsulation
  -Capabilities (JACK, Jadex, ...)
  -Roles (Dastani et al.)
  -Mini-agents (?)

- Recently done: **meta-events**

- To appear soon: annotations for **declarative goals**, improvement in plan failure handling, etc.

- **Debugging** is hard, despite mind inspector, etc.

- Further work on combining with environments and organisations
Summary

- **AgentSpeak**
  - Logic + BDI
  - Agent programming language

- **Jason**
  - AgentSpeak interpreter
  - Implements the operational semantics of AgentSpeak
  - Speech-act based communication
  - Highly customisable
  - Useful tools
  - Open source
  - Open issues
Further Resources


- R.H. Bordini, J.F. Hübner, and M. Wooldrige
  Programming Multi-Agent Systems in AgentSpeak using Jason
Environment Oriented Programming — EOP —
Outline

3. Environment Programming
   - Why Environment Programming in MAS
   - Basic Level
   - Advanced Level
   - A&A and CArtAgO
   - Conclusions and Wrap-up
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

Back to the Notion of Environment in MAS
**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
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
Environment Programming: General Issues

- Defining the interface
  - actions, perceptions
  - data-model

- Defining the environment computational model & architecture
  - how the environment works
  - structure, behaviour, topology
  - core aspects to face: concurrency, distribution

- Defining the environment programming model
  - how to program the environment
Basic Level Overview

MAS

AGENTS

actions

percepts

SIMULATED WORLD

OR

INTERFACE

OR

WRAPPER TO EXISTING TECHNOLOGY

REAL WORLD (PHYSICAL OR COMPUTATIONAL)

EXTERNAL WORLD (PHYSICAL OR COMPUTATIONAL)

Example: JAVA PLATFORM

mimicking
Basic Level: Features

- Environment conceptually conceived as a single monolithic block
  - providing actions, generating percepts

- Environment API
  - to define the set of actions and program actions computational behaviour
    - which include the generation of percepts
  - typically implemented using as single object/class in OO such as Java
    - method to execute actions
    - fields to store the environment state
  - available in many agent programming languages/frameworks
    - e.g., Jason, 2APL, GOAL, JADEX
An Example: *Jason* [Bordini et al., 2007a]

- **Flexible Java-based Environment API**
  - Environment base class to be specialised
    - `executeAction` method to specify action semantics
    - `addPercept` to generate percepts

```java
Environment

- globalPercepts: List<Literal>
- agPercepts: Map<String, List<Literal>>

+init(String[] args)
+stop()

+getPercepts(String agName): List<Literal>
+executeAction(String agName, Structure action): boolean

+addPercept(String agName, Literal p)
+removePercept(String agName, Literal p)

...
```

```java
UserEnvironment

+init(String[] args)
+executeAction(String agName, Structure action): boolean
```

Diagram:
- User Environment
- Agent Architecture
  - `getPercepts`
  - `executeAction`
  - `change percepts`
public class MarsEnv extends Environment {
    private MarsModel model;
    private MarsView view;

    public void init(String[] args) {
        model = new MarsModel();
        view = new MarsView(model);
        model.setView(view);
        updatePercepts();
    }

    public boolean executeAction(String ag, Structure action) {
        String func = action.getFunctor();
        if (func.equals("next")) {
            model.nextSlot();
        } else if (func.equals("move_towards")) {
            int x = (int)((NumberTerm)action.getTerm(0)).solve();
            int y = (int)((NumberTerm)action.getTerm(1)).solve();
            model.moveTowards(x, y);
        } else if (func.equals("pick")) {
            model.pickGarb();
        } else if (func.equals("drop")) {
            model.dropGarb();
        } else if (func.equals("burn")) {
            model.burnGarb();
        } else {
            return false;
        }
        updatePercepts();
        return true;
    }

    ...
// mars robot 1

/* Initial beliefs */
at(P) :- pos(P,X,Y) & pos(r1,X,Y).

/* Initial goal */
!check(slots).

/* Plans */

+!check(slots) : not garbage(r1) <- next(slot);
    !check(slots).
+!check(slots).
+!check(slots).

+garbage(r1) : not .desire(carry_to(r2))
     <- !carry_to(r2).

+!carry_to(R)
     <- // remember where to go back
         ?pos(r1,X,Y);
         +++pos(last,X,Y);

     // carry garbage to r2
     !take(garb,R);

     // goes back and continue to check
     !at(last);
     !!check(slots).

...
Another Example: 2APL [Dastani, 2008b]

- 2APL
  - BDI-based agent-oriented programming language integrating declarative programming constructs (beliefs, goals) and imperative style programming constructs (events, plans)

- Java-based Environment API
  - Environment base class
  - implementing actions as methods
    - inside action methods external events can be generated to be perceived by agents as percepts
Example: Block-world Environment in 2APL

```java
package blockworld;

public class Env extends apapl.Environment {

    public void enter(String agent, Term x, Term y, Term c){...}

    public Term sensePosition(String agent){...}

    public Term pickup(String agent){...}

    public void north(String agent){...}

    ...

}
```
BeliefUpdates:
  \{ \text{bom}(X,Y) \} \quad \text{RemoveBomb}(X,Y) \{ \text{not bom}(X,Y) \}
  \{ \text{true} \} \quad \text{AddBomb}(X,Y) \{ \text{bom}(X,Y) \}
  \{ \text{carry(bom)} \} \quad \text{Drop}( ) \quad \{ \text{not carry(bom)} \}
  \{ \text{not carry(bom)} \} \quad \text{PickUp}( ) \quad \{ \text{carry(bom)} \}

Beliefs:
  \text{start}(0,1).
  \text{bom}(3,3).
  \text{clean}( \text{blockW} ) :=
    \quad \text{not bom}(X,Y) \& \text{not carry(bom)}.

Plans:
  \text{B}\{\text{start}(X,Y)\} ;
  \text{@blockworld}\{ \text{enter}( X, Y, \text{blue} ), L \}

Goals:
  \text{clean}( \text{blockW} )

PG-rules:
  \text{clean}( \text{blockW} ) \leftarrow \text{bom}( X, Y ) |$
  \{ \quad \text{goto}( X, Y ) ;$
    \text{@blockworld}\{ \text{pickup}( ), L1 \} ;$
    \text{PickUp}( ) ;$
    \text{RemoveBomb}( X, Y ) ;$
    \text{goto}( 0, 0 ) ;$
    \text{@blockworld}\{ \text{drop}( ), L2 \} ;$
    \text{Drop}( )$
  \}$

PC-rules:
  \text{goto}( X, Y ) \leftarrow \text{true} |$
  \{ \text{@blockworld}\{ \text{sensePosition}(), \text{POS} \};$
    \text{B}(\text{POS} = [A,B]) ;$
    \text{if} B(\text{A} > X) \text{ then}$
    \{ \text{@blockworld}\{ \text{west}(), L \} ;$
      \text{goto}( X, Y )$
    \}$
    \text{else if} B(\text{A} < X) \text{ then}$
    \{ \text{@blockworld}\{ \text{east}(), L \} ;$
      \text{goto}( X, Y )$
    \}$
    \text{else if} B(\text{B} > Y) \text{ then}$
    \{ \text{@blockworld}\{ \text{north}(), L \} ;$
      \text{goto}( X, Y )$
    \}$
    \text{else if} B(\text{B} < Y) \text{ then}$
    \{ \text{@blockworld}\{ \text{south}(), L \} ;$
      \text{goto}( X, Y )$
    \}$
  \}$
Recent initiative supported by main APL research groups [Behrens et al., 2010]

- GOAL, 2APL, GOAL, JADEX, JASON

Goal of the initiative
- design and develop a generic environment interface standard
  - a standard to connect agents to environments
  - ... environments such as agent testbeds, commercial applications, video games...

Principles
- wrapping already existing environments
- creating new environments by connecting already existing apps
- creating new environments from scratch

Requirements
- generic
- reuse
EIS Meta-Model

- By means of the Env. Interface agents perform actions and collect percepts
  - actually actions/percepts are issued to controllable entities in environment model
  - represent the agent bodies, with effectors and sensors
Interface functions
- attaching, detaching, and notifying observers (software design pattern);
- registering and unregistering agents;
- adding and removing entities;
- managing the agents-entities-relation;
- performing actions and retrieving percepts;
- managing the environment

Interface Intermediate language
- to facilitate data-exchange
- encoding percepts, actions, events
Vision: environment as a **first-class abstraction** in MAS [Weyns et al., 2007, Ricci et al., 2010b]

- **application** or **endogenous** environments, i.e. that environment which is an explicit part of the MAS
- providing an exploitable **design & programming** abstraction to build MAS applications

Outcome

- distinguishing clearly between the responsibilities of agent and environment
  - separation of concerns
- improving the engineering practice
Three Support Levels [Weyns et al., 2007]

- Basic **interface** support
- **Abstraction** support level
- **Interaction-mediation** support level
Basic Interface Support

- The environment enables agents to access the deployment context
  - i.e. the hardware and software and external resources with which the MAS interacts
Abstraction Support

- Bridges the conceptual gap between the agent abstraction and low-level details of the deployment context
  - shields low-level details of the deployment context
• **Regulate** the access to shared resources
• **Mediate** interaction between agents
Environment definition revised [Weyns et al., 2007]

The environment is a first-class abstraction that provides the surrounding conditions for agents to exist and that mediates both the interaction among agents and the access to resources.
Environments for Multi-Agent Systems research field / **E4MAS** workshop series [Weyns et al., 2005]
- different themes and issues (see JAAMAS Special Issue [Weyns and Parunak, 2007] for a good survey)
  - mechanisms, architectures, infrastructures, applications [Platon et al., 2007, Weyns and Holvoet, 2007, Weyns and Holvoet, 2004, Viroli et al., 2007]
  - the main perspective is (agent-oriented) software engineering

Focus of this tutorial: the role of the environment abstraction in **MAS programming**
- environment programming
Environment Programming

- Environment as **first-class programming abstraction** [Ricci et al., 2010b]
  - software designers and engineers perspective
  - endogenous environments (vs. exogenous one)
  - programming MAS = programming Agents + programming Environment
    - ..but this will be extended to include OOP in next part

- Environment as **first-class runtime abstraction** for agents
  - agent perspective
  - to be observed, used, adapted, constructed, ...

- Defining computational and programming frameworks/models also for the environment part
Computational Frameworks for Environment Programming: Issues

- Defining the environment interface
  - actions, percepts, data model
  - **contract** concept, as defined in software engineering contexts (Design by Contract)

- Defining the environment computational model
  - environment structure, behaviour

- Defining the environment distribution model
  - topology
Programming Models for the Environment: Desiderata

- **Abstraction**
  - keeping the agent abstraction level e.g. no agents sharing and calling OO objects
  - effective programming models for controllable and observable computational entities

- **Modularity**
  - away from the monolithic and centralised view

- **Orthogonality**
  - wrt agent models, architectures, platforms
  - support for heterogeneous systems
Programming Models for the Environment: Desiderata

- **Dynamic extensibility**
  - dynamic construction, replacement, extension of environment parts
  - support for open systems

- **Reusability**
  - reuse of environment parts for different kinds of applications
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, 2000a]

- **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., 2010b]**
  - Introducing a computational notion of artifact to design and implement agent environments
Agents and Artifacts (A&A) Conceptual Model: Background Human Metaphor

agents can join dynamically the workspace
Agents
- autonomous, goal-oriented pro-active entities
- create and co-use artifacts for supporting their activities
  - besides direct communication

Artifacts
- non-autonomous, function-oriented, stateful entities
  - controllable and observable
  - modelling the tools and resources used by agents
  - designed by MAS programmers

Workspaces
- grouping agents & artifacts
- defining the topology of the computational environment
A&A Programming Model Features [Ricci et al., 2007b]

- Abstraction
  - artifacts as first-class resources and tools for agents
- Modularisation
  - artifacts as modules encapsulating functionalities, organized in workspaces
- Extensibility and openness
  - artifacts can be created and destroyed at runtime by agents
- Reusability
  - artifacts (types) as reusable entities, for setting up different kinds of environments
A&A Meta-Model in More Detail [Ricci et al., 2010b]
Artifact Abstract Representation

- **SIGNALS**
- **USAGE INTERFACE**
- **LINK INTERFACE**
- **OBSERVABLE PROPERTIES**
- **OPERATIONS**

```
OperationX(Params)
...
ObsPropName(Args)
...
SIGNALS
USAGE
INTERFACE
OBSERVABLE
PROPERTIES
OperationY(Params)
...
LINK
INTERFACE
OPERATIONS
```
A World of Artifacts

- count 5
  - inc
  - reset
  a counter

- state true
  - switch
  a flag

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

- next_todo check plant
  - last_todo ...
  a agenda

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

- n_records 1001
  - table_names ...
  a database

- clearEvents
- postEvent
- registerForEvs
  an event service

- out
- in
- rd
  a tuple space
A Simple Taxonomy

- Individual or personal artifacts
  - designed to provide functionalities for a single agent use
    - e.g. an agenda for managing deadlines, a library...

- Social artifacts
  - designed to provide functionalities for structuring and managing the interaction in a MAS
  - coordination artifacts [Omicini et al., 2004], organisation artifacts, ...
    - e.g. a blackboard, a game-board,...

- Boundary artifacts
  - to represent external resources/services
    - e.g. a printer, a Web Service
  - to represent devices enabling I/O with users
    - e.g. GUI, console, etc.
Actions and Percepts in Artifact-Based Environments

- Explicit semantics defined by the (endogenous) environment [Ricci et al., 2010c]
  - success/failure semantics, execution semantics
  - defining the **contract** (in the SE acceptation) provided by the environment

**actions \leftrightarrow \text{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 \leftrightarrow \text{artifacts’ observable properties} + \text{signals}**

properties represent percepts about the state of the environment

signals represent percepts concerning events signalled by the environment
Performing an action corresponds to triggering the execution of an operation

- acting on artifact’s usage interface
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
Basic mechanism to enable inter-artifact interaction

- **linking** artifacts through interfaces (link interfaces)
  - operations triggered by an artifact over an other artifact
- Useful to design & program distributed environments
  - realised by set of artifacts linked together
  - possibly hosted in different workspaces
Artifact Manual

- Agent-readable description of artifact’s...
  - **functionality**
    - **what** functions/services artifacts of that type provide
  - **operating instructions**
    - **how** to use artifacts of that type
- Towards advanced use of artifacts by intelligent agents [Piunti et al., 2008]
  - dynamically choosing which artifacts to use to accomplish their tasks and how to use them
  - strong link with Semantic Web research issues
- Work in progress
  - defining ontologies and languages for describing the manuals
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., 2007c]
  - 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
Integration with existing agent platforms [Ricci et al., 2008]
- by means of bridges creating an action/perception interface and doing data binding

Outcome
- developing open and heterogenous MAS
- introducing a further perspective on interoperability besides the ACL’s one
  - sharing and working in a common work environment
  - common object-oriented data-model
JaCa Platform

Integration of CArtAgO with Jason language/platform
- a JaCa program is a dynamic set of Jason agents working together in one or multiple CArtAgO workspaces

Mapping
- actions
  - Jason agent external actions are mapped onto artifacts’ operations
- percepts
  - artifacts’ observable properties are mapped onto agent beliefs
  - artifacts’ signals are mapped as percepts related to observable events
- data-model
  - Jason data-model is extended to manage also (Java) objects
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)**

```prolog
!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)**

```prolog
!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
Pre-defined Artifacts

- Each workspace contains by default a predefined set of artifacts
  - providing core and auxiliary functionalities
  - i.e. a pre-defined repertoire of actions available to agents...
- Among the others
  - workspace, type: cartago.WorkspaceArtifact
    - functionalities to manage the workspace, including security
    - operations: makeArtifact, lookupArtifact, focus,…
  - node, type: cartago.NodeArtifact
    - core functionalities related to a node
    - operations: createWorkspace, joinWorkspace, …
  - console, type cartago.tools.Console
    - operations: println,…
  - blackboard, type cartago.tools.TupleSpace
    - operations: out, in, rd, …
- ….
Example 2: Coordination Artifacts – A Bounded Buffer

```java
public class BoundedBuffer extends Artifact {
    private LinkedList<Item> items;
    private int nmax;

    void init(int nmax){
        items = new LinkedList<Item>();
        defineObsProperty("n_items",0);
        this.nmax = nmax;
    }

    @OPERATION void put(Item obj){
        await("bufferNotFull");
        items.add(obj);
        getObsProperty("n_items").updateValue(items.size());
    }

    @OPERATION void get(OpFeedbackParam<Item> res) {
        await("itemAvailable");
        Item item = items.removeFirst();
        res.set(item);
        getObsProperty("n_items").updateValue(items.size());
    }

    @GUARD boolean itemAvailable(){ return items.size() > 0; }

    @GUARD boolean bufferNotFull(Item obj){ return items.size() < nmax; }
}
```

- Basic operation features
  - output parameters to represent action feedbacks
  - long-term operations, with a high-level support for synchronization (await primitive, guards)
Example 2: Producers and Consumers

**PRODUCERS**

item_to_produce(0).
!produce.

+!produce: true
  <- !setupTools(Buffer);
  !produceItems.

+!produceItems : true
  <- ?nextItemToProduce(Item);
  put(Item);
  !produceItems.

+?nextItemToProduce(N) : true
  <- -item_to_produce(N);
  +item_to_produce(N+1).

+!setupTools(Buffer) : true
  <- makeArtifact("myBuffer","BoundedBuffer", [10],Buffer).

-!setupTools(Buffer) : true
  <- lookupArtifact("myBuffer",Buffer).

**CONSUMERS**

!consume.

+!consume: true
  <- ?bufferReady;
  +!consumeItems.

+!consumeItems: true
  <- get(Item);
  !consumeItem(Item);
  !consumeItems.

+!consumeItem(Item) : true
  <- .my_name(Me);
  println(Me,"":",Item).

+?bufferReady : true
  <- lookupArtifact("myBuffer",_).
-?bufferReady : true
  <-.wait(50);
  ?bufferReady.
Remarks

- Process-based operation execution semantics
  - action/operation execution can be long-term
  - action/operation execution can overlap
  - key feature for implementing coordination functionalities
- Operation with output parameters as action feedbacks
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
Example 3: Internal Processes – A Clock

- **Internal operations**
  - execution of operations triggered by other operations
  - implementing controllable *processes*
Example 4: Artifacts for User I/O – GUI Artifacts

- Exploiting artifacts to enable interaction between human users and agents
Example 4: Agent and User Interaction

**GUI ARTIFACT**

```java
public class MySimpleGUI extends GUIArtifact {
  private MyFrame frame;

  public void setup() {
    frame = new MyFrame();
    linkActionEventToOp(frame.okButton, "ok");
    linkKeyStrokeToOp(frame.text, "ENTER", "updateText");
    linkWindowClosingEventToOp(frame, "closed");
    defineObsProperty("value", getValue());
    frame.setVisible(true);
  }

  @INTERNAL_OPERATION void ok(ActionEvent ev) {
    signal("ok");
  }

  @OPERATION void setValue(double value) {
    frame.setText("" + value);
    updateObsProperty("value", value);
  }

  @INTERNAL_OPERATION void updateText(ActionEvent ev) {
    updateObsProperty("value", getValue());
  }

  private int getValue() {
    return Integer.parseInt(frame.getText());
  }
}
```

**USER ASSISTANT AGENT**

```java
!test_gui.
+!test_gui <- makeArtifact("gui", "MySimpleGUI", Id);
  focus(Id).

+value(V) <- println("Value updated: ", V).
+ok : value(V) <- setValue(V + 1).
+closed <- .my_name(Me);
  .kill_agent(Me).
```
Other Features

- Other CArtAgO features not discussed in this lecture
  - linkability
    - executing chains of operations across multiple artifacts
  - multiple workspaces
    - agents can join and work in multiple workspaces, concurrently
    - including remote workspaces
  - RBAC security model
    - workspace artifact provides operations to set/change the access control policies of the workspace, depending on the agent role
    - ruling agents’ access and use of artifacts of the workspace
  - ...

- See CArtAgO papers and manuals for more information
A&A and CArtAgO: Some Research Explorations

- Designing and implementing artifact-based organisation Infrastructures
  - JaCaMo model and platform (which is the evolution of the ORA4MAS infrastructure [Hübner et al., 2009b])

- Cognitive stigmergy based on artifact environments [Ricci et al., 2007a]
  - cognitive artifacts for knowledge representation and coordination [Piunti and Ricci, 2009]

- Artifact-based environments for argumentation [Oliva et al., 2010]

- Including A&A in AOSE methodology [Molesini et al., 2005]

- Defining a Semantic (OWL-based) description of artifact environments (CArtAgO-DL)
  - JaSa project = JASDL + CArtAgO-DL

- ...
Applying CArtAgO and JaCa

- Using CArtAgO/JaCa for building real-world applications and infrastructures
- Some examples
  - JaCa-Android
    - implementing mobile computing applications on top of the Android platform using JaCa [Santi et al., 2011]
  - JaCa-WS / CArtAgO-WS
    - building SOA/Web Services applications using JaCa [Ricci et al., 2010a]
  - JaCa-Web
    - implementing Web 2.0 applications using JaCa
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
  - JaCa case
Organisation Oriented Programming — OOP —
Abstractions in Multi-Agent Systems

Diagram showing layers of abstraction:
- **Organisation Level**: 
  - `org`
  - `schema`
  - `role`
  - `mission`
- **Agent Level**: 
  - `agent`
  - `artifact`
  - `wsp`
- **Endogenous Environment Level**: 
  - `network node`
- **Exogenous Environment Level**: 
  - `node`
Outline

4 Organisation Oriented Programming (OOP)
  - Fundamentals
  - Motivations
  - Some OOP approaches
  - Focus on the Moise framework
    - Moise Organisation Modelling Language (OML)
    - Moise Organisation Management Infrastructure (OMI)
Organisation in MAS – a definition

- What is an organisation?
Organisation in MAS – a definition

- What is an organisation?

- Pattern of agent cooperation
  - with a purpose
  - supra-agent
  - emergent or
  - predefined (by designer or agents)
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**
Introduction

Motivations

OOP

Moise

Conclusion

Perspective on organisations from EASSS’05 Tutorial (Sichman, Boissier)
Perspective on organisations from EASSS’05 Tutorial (Sichman, Boissier)

- **Introduction**
  - Motivations

- **OOP**

- **Conclusion**

**Agent Centred**
- Swarms, AMAS, SASO
- Self-organisations …
- Organisation is observed.
- Implicitly programmed in Agents, Interactions, Environment.

**Social Reasoning**
- Coalition formation
- Contract Net Protocol …
- Organisation is observed.
- Coalition formation mechanisms programmed in Agents.

**Organisation Centred**
- AOSE
- MASE, GAIA, MESSAGE, …
- Organisation is a design model.
- It is hard-coded in Agents

**Organisation-Oriented Programming of MAS**
- TAEMS, STEAM, AGR
- MOISE+, OPERA, …
- Organisation is observed.
- Coalition formation mechanisms programmed in Agents.

**Agents don’t know about organisation**

**Agents know about organisation**

**Organisation Specification**

**Organisation Entity**

**Local Representation**

**Observed Organisation**

**Designer / Observer**

**Bottom-up**

**Top-down**
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
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 environment
Components of OOP: Organisation Modelling Language (OML)

- Declarative specification of the organisation(s)
- Specific constraints, norms and cooperation patterns imposed on the agents
- Based on an organisational model
  - e.g. AGR [Ferber and Gutknecht, 1998],
    TeamCore [Tambe, 1997],
    Islander [Esteva et al., 2001],
    Moise+ [Hübner et al., 2002],
    Opera [Dignum and Aldewereld, 2010],
    2OPL [Dastani et al., 2009],
    ...

Components of OOP:
Organisation Modelling Language (OML)
Components of OOP: Organisation Management Infrastructure (OMI)

- **Coordination mechanisms**, i.e. support infrastructure
  - e.g. MadKit [Gutknecht and Ferber, 2000b], Karma [Pynadath and Tambe, 2003], ...

- **Regulation mechanisms**, i.e. governance infrastructure
  - e.g. Ameli [Esteva et al., 2004], S-Moise+ [Hübner et al., 2006], ORA4MAS [Hübner et al., 2009a], ...

- **Adaptation mechanisms**, i.e. reorganisation infrastructure
Components of OOP: Integration mechanisms

- **Agent** integration mechanisms allow agents to be aware of and to deliberate on:
  - entering/exiting the organisation
  - modification of the organisation
  - obedience/violation of norms
  - sanctioning/rewarding other agents

  e.g. $J$-Moise$^+$ [Hübner et al., 2007], Autonomy based reasoning [Carabelea, 2007], ProsA$_2$ Agent-based reasoning on norms [Ossowski, 1999], ...

- **Environment** integration mechanisms transform organisation into embodied organisation so that:
  - organisation may act on the environment (e.g. enact rules, regimentation)
  - environment may act on the organisation (e.g. count-as rules)

  e.g [Piunti et al., 2009b], [Okuyama et al., 2008]
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: **Constitutive** point of view

- Organisation **helps** the agents to cooperate with other agents by defining **common** cooperation schemes
  - global tasks
  - protocols
  - groups, responsibilities

**e.g.** ‘to bid’ for a product on eBay is an **institutional action** only possible because eBay defines rules for that very action
  - the bid protocol is a constraint but it also **creates** the action

**e.g.** when a soccer team wants to play match, the organisation helps the members of the team to synchronise actions, to share information, etc
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
Motivations for OOP: Agents point of view

An organisational specification is required to enable agents to “reason” about the organisation:

- to decide to enter into/leave from the organisation during execution
  - → Organisation is no more closed
- to change/adapt the current organisation
  - → Organisation is no more static
- to obey/disobey the organisation
  - → Organisation is no more a regimentation
Motivations for OOP:

**Organisation point of view**

An organisational specification is required to enable the organisation to “reason” about itself and about the agents in order to ensure the achievement of its global purpose:

- to decide to let agents enter into/leave from the organisation during execution
  - → Organisation is no more closed

- to decide to let agents change/adapt the current organisation
  - → Organisation is no more static and blind

- to govern agents behaviour in the organisation (i.e. monitor, enforce, regiment)
  - → Organisation is no more a regimentation
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]
- ...
Summary

- Several models
- Several dimensions on modelling organisation
  - Structural (roles, groups, ...)
  - Functional (global plans, ....)
  - Dialogical (scenes, protocols, ...)
  - Normative (norms)
The Moise Framework
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., 2009a] 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., 2009a])
  - Agents and Organisation (J-Moise+ [Hübner et al., 2007])
Moise Modelling Dimensions

Introduction

Motivations

OOP

Environment

O₂

P

Functional
 Specification

Global goals, plans, Missions, schemas, preferences

B

Structural
 Specification

Groups, links, roles

Compatibilities, multiplicities

inheritance

Normative Specification

Permissions, Obligations

Allows agents autonomy!
Moise OML meta-model (partial view)
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
  - \(\mathcal{J}\)-Moise\(^+\) [Hübner et al., 2007]
- the Organisation Management Infrastructure
  - ORA4MAS [Hübner et al., 2009a]
Moise OML 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
Moise OML 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
Functional Specification Example

Graphical representation of social scheme “side_attack” for joj team

Key

Organizational Entity

Lucio \( m_1 \)

Cafu \( m_2 \)

Rivaldo \( m_3 \)
Goal States

- **waiting**: initial state
- **enabled**: goal pre-conditions are satisfied & scheme is well-formed
- **satisfied**: agents committed to the goal have achieved it
- **impossible**: the goal is impossible to be satisfied
Moise OML 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 and links
- Normative roles
- Functional schemas and missions
- Purpose

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

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, $S$-$Moise^+$, ...)
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
Programming MAS in JaCaMo
Putting the Pieces Together
Exploiting Orthogonality

- Treating AOP & EOP & OOP as **orthogonal** dimensions
  - improving **separation of concerns**
    - using the best abstraction level and tools to tackle the specific dimensions, avoiding design pitfalls, such as using agents to implement either non-autonomous entities (e.g., a blackboard agent) or a collection of autonomous entities (group agent)
  - promoting openness and heterogeneity
    - E.g., heterogeneous agents working in the same organisation, heterogeneous agents working in the same environment, the same agent working in different and heterogeneous organisations, the same agent working in different heterogeneous environments

- Outcome from a programming point of view
  - code more clean and understandable
  - improving modularity, extensibility, reusability
A single platform/programming model integrating
- *Jason* (AOP)
- CArtAgO) (EOP)
- Moise (OOP)

A *JaCaMo* system is composed by
- a dynamic set of BDI agents programmed in *Jason*
- working inside artifact-based environments programmed in CArtAgO
- organized in terms of Moise organizations
JaCaMo Meta-Model

Environment dimension

- Workspace
- Artifact
- Operation
- Observable Property
- Observable Event

Agent dimension

- Action
- Plan
- Trigger event
- Belief
- Goal

Organisation dimension

- Group
- Role
- Norm
- Social Scheme
- Mission

Legend

- composition
- association
- dependency
- agent’s actions
- dimension border
- concept mapping
Downloads

- **JaCaMo** within the eclipse platform
  - Jason \(\geq 1.3.7\) https://jason.sf.net
  - Jason Eclipse Plugin
    https://docs.google.com/document/pub?id=1URDYMtFP64rHnlb7GcnKyUGbaypNcZmteWupWD5p6sM

- **Other Relevant URLs and Projects**
  - **JaCaMo** http://jacamo.sourceforge.net
  - **Jason** http://jason.sourceforge.net
  - **CArtAgO** http://cartago.sourceforge.net
  - **Moise** http://moise.sourceforge.net
  - http://jason.sourceforge.net/jBook
JaCaMo Wizard

Select a wizard
Create a Jason Project

Wizards:
- General
- CVS
- Eclipse Modeling Framework
- Git
- Jason
  - Agent
  - CArtAgO Artifact
  - Internal Action
- Jason Project
- Java
- Maven
- Plug-in Development
- Server
- Tasks
Configuring the Infrastructure

New Jason Project
This wizard creates a new Jason Project

Project name: easss
Infrastructure: JaCaMo
Environment: Without environment
Code organization
Running an *JaCaMo* Application
Configuration

Application in execution

- Plumbing OK
- Electrical System OK
- Exterior Painting OK
- Interior Painting OK

Roof OK

Walls OK

Windows OK

Doors OK

Floor OK

OK

Responsible groups: hsh_group.

Players
- companyA committed to install_plumbing
- companyB2 committed to paint_house
- companyC2 committed to install_electrical_system
- companyD5 committed to prepare_site
- companyD6 committed to fit_doors
- companyD8 committed to fit_windows
- companyE committed to build_walls
- companyE committed to build_roof
- companyE committed to lay_floors
- giacomo committed to management_of_house_building

goal | state | committed/achieved by | arguments | plan
--- | --- | --- | --- | ---
house_built | satisfied | [giacomo](#) | [giacomo](#) | site_prepared floors_laid walls_built rwd pee interior_painted
site_prepared | satisfied | [companyD5](#) | [companyD5](#) |
floors_laid | satisfied | [companyE](#) | [companyE](#) |
walls_built | satisfied | [companyE](#) | [companyE](#) |
rwd | satisfied | [V](#) | | windows_fitted || doors_fitted
Bibliography I


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


Normative multi-agent programs and their logics.


Operetta: Organization-oriented development environment.


On the formal specification of electronic institutions.


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

Formal semantics for an abstract agent programming language.

*Journal of Autonomous Agents and Multi-Agent Systems.*

*Autonomous Agents and Multi-Agent Systems.* 
DOI-URL: http://dx.doi.org/10.1007/s10458-009-9084-y.


Agentspeak(l): Bdi agents speak out in a logical computable language.  
In de Velde, W. V. and Perram, J. W., editors, MAAMAW, volume 1038 of  

A platform for developing SOA/WS applications as open and heterogeneous  
multi-agent systems. 

Cognitive stigmergy: Towards a framework based on agents and artifacts.  
In Weyns, D., Parunak, H. V. D., and Michel, F., editors, Environments for  
Integrating artifact-based environments with heterogeneous agent-programming platforms.
In *Proceedings of 7th International Conference on Agents and Multi Agents Systems (AAMAS08)*.

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).

Environment programming in CArtAgO.


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