TU Clausthal
Clausthal University of Technology

PUCRS

# Chapter 13: Programming Multi-Agent Systems

Rafael H. Bordini and Jürgen Dix

Multi-Agent Systems, edited by Gerhard Weiss
MIT Press, May 2012

## Time

Duration: The course can be divided into 4 lectures à 60 minutes:

## Course type

Level: advanced
Prerequisites:

## Course website

http://mitpress.mit.edu/multiagentsystems

# Course Overview

The course can be divided into 4 lectures à 60 minutes:

- Lec. 1: History and the MAOP Paradigm
- Lec. 2: Examples of Programming Languages
- Lec. 3: Organisation and Environment Programming
- Lec. 4: An Example in JaCoMo

# Reading Material I

Rafael Bordini and Jürgen Dix (2012).
Chapter 13: Programming Multi-agent Systems.
In G. Weiss (Ed.), Multiagent Systems, MIT Press.

Rafael H. Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah-Seghrouchni, editors.
*Multi-agent Programming: Languages, Tools and Applications*.
Springer, 2009.

# Reading Material II

Rafael H. Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah-Seghrouchni, editors,
*Multi-agent Programming: Languages, Platforms and Applications*.
Springer, 2005.

Rafael H. Bordini, Jomi F. Hübner, and Michael Wooldridge.
*Programming Multi-Agent Systems in AgentSpeak Using Jason*.
Wiley, 2007.

# Outline

1 History and the MAOP Paradigm

2 Examples of Programming Languages

3 Organisation and Environment Programming

4 An Example in JaCaMo

5 References

# 1. History and the MAOP Paradigm

1 History and the MAOP Paradigm
- Agent Level
- Environment Level
- Social Level

# Agent0

- Agent-oriented programming started with Shoham in 1993.
- While the first decade saw mainly theoretical approaches, the creation of the ProMAS and DALT workshop series (both held with AAMAS since 2003) and related activity helped to change the picture.
- The first agent programming languages were mostly concerned with programming **individual** agents: no abstractions covering the social and environmental dimensions.

# Mature Languages

- Usable **IDEs** and **debugging tools** (in particular tools to inspect the state of an agent or an organisation).
- Still a long way to go compared to the best tools used for object-oriented programming.
- Inspiration comes from **reactive planning systems** [Georgeff and Lansky, 1987] and the *societal view* of computing.

# Ongoing research

- Proceedings of ProMAS [Collier et al., 2011],
- proceedings of DALT [Omicini et al., 2011],
- CLIMA, AAMAS (as well as the main AI conferences),
- LADS [Dastani et al., 2010] and various other workshops.

# Survey papers

- [Fisher et al., 2007, Bordini et al., 2006, Mascardi et al., 2004, Dastani and Gómez-Sanz, 2005]
- [Bordini et al., 2011, Bordini et al., 2007a].

# Reacting to Events × Long-Term Goals

- Auonomous agents have to be **attentive** to changes and react to them appropriately as former goals may not succeed.
- *Long-term goals* have to be taken into account.
- In **highly dynamic environments**, not reacting to events means losing opportunities for the agent to achieve what is expected of it.

# Courses of Action Depend on Circumstances

- Agents will be **constantly deciding** which courses of action to take in order to react to events.
- This decision depends on the **current circumstances** (of the agent, other agents, the environment, etc.).
- The agent will use its most up-to-date information about the state of itself, other agents, and the environment in order to **decide at runtime** what needs to be done.

# Choosing Courses of Action only When About to Act

- Due to the highly dynamic nature, the course of action to be used should **not be decided too early**: things might have changed by the time the agent is actually about to act.
- Agent languages often use partially instantiated plans so that not only details of a plan but the particular (sub)plan to be used for each (sub)goal is only chosen when the agent is about to act on achieving a particular goal.

# Dealing with Plan Failure

- Even **delaying the decision** on particular courses of action might not be enough to ensure that the agent has chosen a suitable course of action in a dynamic environment.
- While **executing** a plan, the agent may realise a failure has occurred, so agent languages still need to provide mechanisms to deal with plan failure.

# Rational Behaviour

- Agent applications will require that agents behave rationally.
- BDI literature [Rao and Georgeff, 1995] has pointed to very concrete **aspects of rationality**.
- If an agent has an intention (i.e. is committed to the goal of achieving a particular state of affairs) we expect it to reason about how to achieve that intention.
- We do not expect the agent to give up before the intention is believed to have been effectively achieved, unless there is good reason to believe it will not be possible to achieve it at all.

# Social Ability – High-Level Communication, Organisation

- Essential feature: some tasks are only possible if agents interact.
- In order to cooperate or to coordinate their action, agents typically use a **high-level form of communication** based on the idea of speech-acts [Austin, 1962, Searle, 1969].
- Agents can be programmed to take part in an agent organisation all within the context of multiagent oriented programming.

# Code Modification at Runtime

- Platforms for MAP allow for simple changing the system program at runtime.
- Plan libraries can be changed at runtime, and so does the behaviour of the agent.
- Often this is done through speech-act based communication: not only other agents but humans as well can communicate new plans (i.e. know-how or behaviour) for the agents.
- In some platforms for agent organisations the specification of the social structure and overall social plan and norms that agents ought to follow can be changed on-the-fly.

# 1.1 Agent Level

# General Abstractions

- MAOP provides abstractions to facilitate the development of software that is both **autonomous** and **social**.
- belief is an abstraction of the agent's informational state .
- Agents need to be able not just to represent beliefs but to continuously update them.
- Perhaps the most important abstraction in agent programming is that of a goal.
- A goal is typically represented as a **property that is currently not believed to be true** and that will lead the agent into action in order to make that property true

# Declarative achievement goal

- The agent wishes to bring about a certain state of affairs which it currently believes not to hold and is willing to commit itself to acting so as to bring about such state of affairs: Declarative achievement goal.
- Such goals facilitate the programming of software that can appear to be pro-active as well as recovering from failure due to a quickly changing environment.
- One of the first comprehensive typologies for goals in agent programming was published in [Braubach et al., 2004], with much work following it, in [van Riemsdijk et al., 2005].

# Plans and Intentions

- A plan is a course of action that under specific circumstances might help the agent handle a particular event (achieving a long-term goal or reacting to changes in beliefs, for example about the environment).
- An intention is an instance of a plan that has been chosen to handle a particular event and has been partially instantiated with information about the event.
- This **intended means** may contain further goals to achieve.
- The agent uses information as up-to-date as possible when committing to particular means to achieve its goals.

# 1.2 Environment Level

- A typical abstraction at the environment level is that of an artifact: a **non-autonomous**, **non-proactive** entity which however is not an object in object orientation.
- An artifact transparently encapsulates two other important abstractions connecting agents and their environment: **actions** and **percepts**.
- Artifacts can be used to transparently give agents access to software services. They can also be used to create a model of a real-world environment.

# 1.3 Social Level

# Organisations, obligations, norms

- An agent organisation typically has a structure, possibly hierarchical, formed by **groups** of agents, where individual agents might play specific **roles**.
- If an agent autonomously choses to adopt a specific role in an agent organisation, it will commit to specific obligations that the organisation expect of agents playing that role.
- Such obligations, prohibitions, and permissions are specified by means of social norms.

# Organisations, obligations, norms II

- Norms can be enforced by regimentation, i.e. the system prevents the violation of the norm to even take place, or **sanctions** might be specified so as to punish agents that do not comply with particular norms.
- Social plans can be used to explicitly represent the specific subgoals that each agent in a group is expected to achieve in order for a task that requires the joint work of a team of agents to be accomplished.

# 2. Examples of Programming Languages

## 2 Examples of Programming Languages

- JASON
- Other BDI-Based Languages
- Approaches based on executable logics

# Overview

- We focus here on individual agent programs.
- We present mainly JASON and mention in passing a few other languages.

# 2.1 Jason

# AgentSpeak

- 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

- 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 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

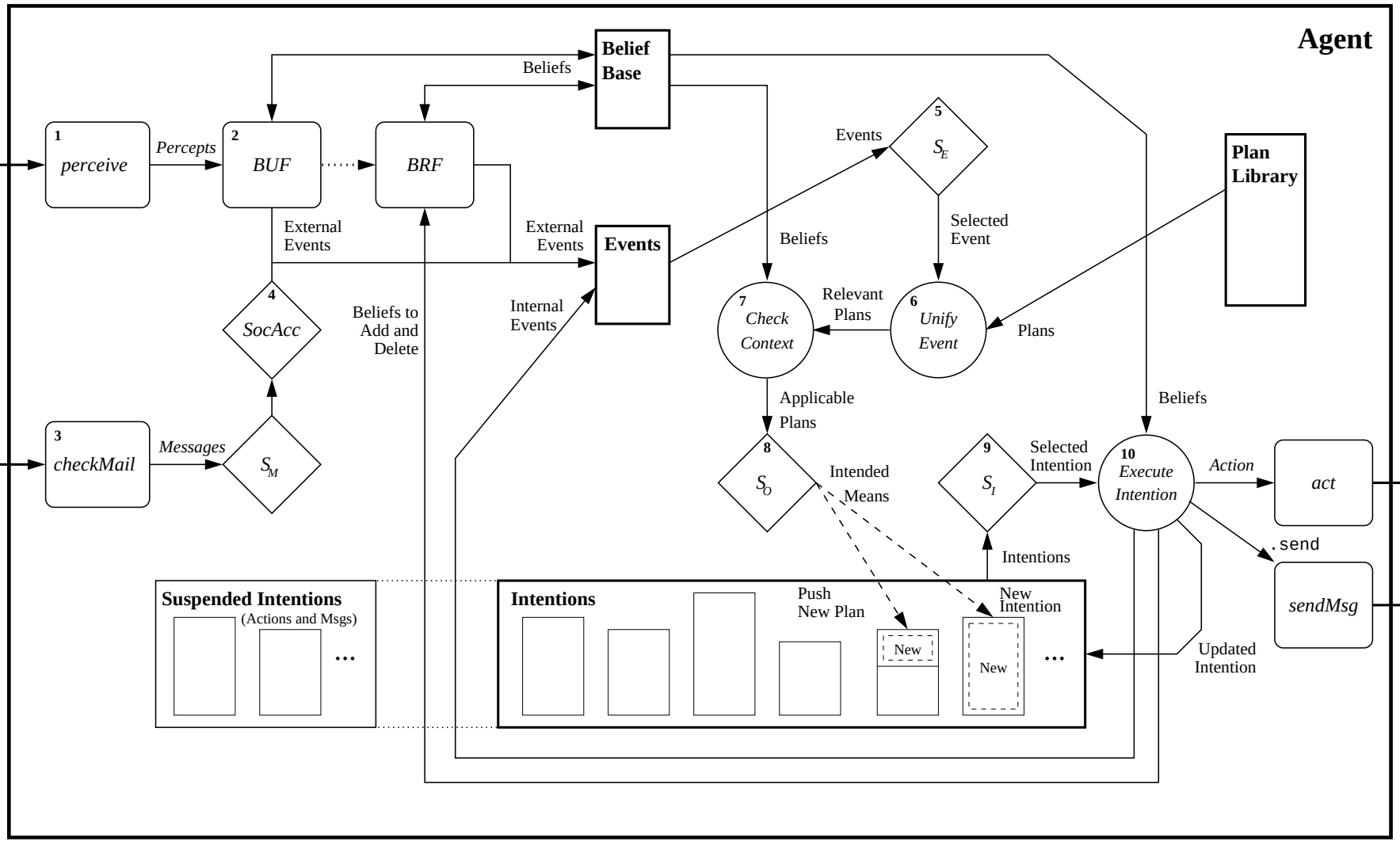
**Plan library:** stores all the plans currently known by the agent

**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

- 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 Rreasoning Cycle



# Beliefs — Representation

## Syntax

Beliefs are represented by annotated literals of first order logic

$\texttt{functor(}term_1\texttt{, ..., }term_n\texttt{)[}annot_1\texttt{, ..., }annot_m\texttt{]}$

## Example 2.1 (belief base of agent Tom)

```
red(box1)[source(percept)].
friend(bob,alice)[source(bob)].
lier(alice)[source(self),source(bob)].
~lier(bob)[source(self)].
```

# Beliefs — Dynamics

## 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)]
```

# Beliefs — Dynamics II

## By communication

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

```
.send(tom,tell,lier(alice)); // sent by bob
// adds lier(alice)[source(bob)] in Tom's BB
...
.send(tom,untell,lier(alice)); // sent by bob
// removes lier(alice)[source(bob)] from Tom's BB
```

# Goals — Representation

## Types of goals

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

## Syntax

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

# Goals — Representation II

## Example 2.2 (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)

```
...
// 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
```

# Triggering Events — Representation

- Events happen as consequence to changes in the agent's beliefs or goals
- An agent reacts to events by executing plans
- Types of plan triggering events

| | |
|---|---|
| +b | (belief addition) |
| -b | (belief deletion) |
| +!g | (achievement-goal addition) |
| -!g | (achievement-goal deletion) |
| +?g | (test-goal addition) |
| -?g | (test-goal deletion) |

# Plans — Representation

An AgentSpeak plan has the following general structure:

triggering_event : context <- body.

where:

- the triggering event denotes the events that the plan is meant to handle
- 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

A plan body may contain:

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

# Plans — Operators for Plan Body II

## Example 2.3 (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

```
+green_patch(Rock)[source(percept)]
   :  not battery_charge(low)
   <- ?location(Rock,Coordinates);
      !at(Coordinates);
      !examine(Rock).

+!at(Coords)
   :  not at(Coords) & safe_path(Coords)
   <- move_towards(Coords);
      !at(Coords).
+!at(Coords)
   :  not at(Coords) & not safe_path(Coords)
   <- ...
+!at(Coords) :  at(Coords).
```

# Plans — Dynamics

The plans that form the plan library of the agent 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 2.4

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

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

# Prolog-like Rules in the Belief Base

## Example 2.5

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

# Plan Annotations II

## Example 2.6 (an annotated plan)

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

# Failure Handling: Contingency Plans

## Example 2.7 (an agent blindly committed to g)

```
+!g :  g.

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

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

# Higher-Order Variables

## Example 2.8 (an agent that asks for plans *on demand*)

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

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

# Higher-Order Variables II

- 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(`$term_1, term_2, \ldots$`)`
  - `.union(`$list_1$`,` $list_2$`,` $list_3$`)`
  - `.my_name(`$var$`)`
  - `.send(`$ag$`,`$perf$`,`$literal$`)`
  - `.intend(`$literal$`)`
  - `.drop_intention(`$literal$`)`
- Many others available for: printing, sorting, list/string operations, manipulating the beliefs/annotations/plan library, creating agents, waiting/generating events, etc.

# Suspending and Resuming Intentions

## Example 2.9 (JASON code with meta-events)

```
+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)).
```

# Communication Infrastructure

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

# Example of an Environment Class

```
1 import jason.*;
2 import ...;
3 public class robotEnv extends Environment {
4   ...
5   public robotEnv() {
6     Literal gp =
7           Literal.parseLiteral("green_patch(souffle)");
8     addPercept(gp);
9   }
10
11  public boolean executeAction(String ag, Structure action) {
12    if (action.equals(...)) {
13      addPercept(ag,
14           Literal.parseLiteral("location(souffle,c(3,4))");
15    }
16    ...
17    return true;
18 } }
```

# MAS Configuration Language

Simple way of defining a multi-agent system

## Example 2.10 (MAS that uses JADE as infrastructure)

```
MAS my_system {
    infrastructure: Jade
    environment: robotEnv
    agents:
         c3po;
         r2d2 at jason.sourceforge.net;
         bob #10; // 10 instances of bob
    classpath: "../lib/graph.jar";
}
```

# MAS Configuration Language II

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

## Example 2.11 (MAS with customised agent)

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

# MAS Configuration Language III

## Example 2.12 (CARTAGO environment)

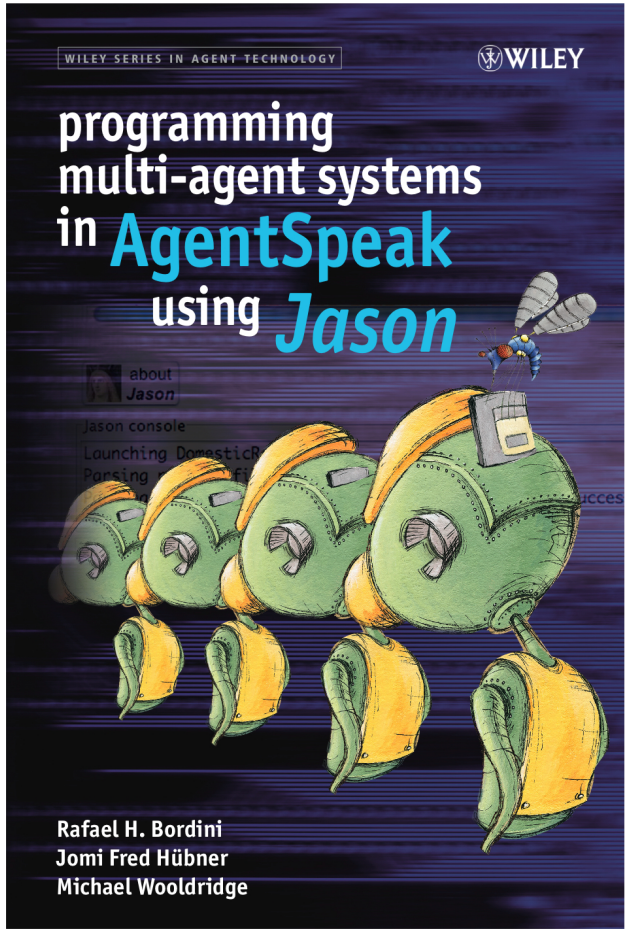
```
MAS grid_world {

  environment: alice.c4jason.CEnv

  agents:
      cleanerAg
          agentArchClass alice.c4jason.CogAgentArch
          #3;
}
```

# JASON Customisations

- Agent class customisation:
  selectMessage, selectEvent, selectOption,
  selectIntetion, buf, brf, ...
- Agent architecture customisation:
  perceive, act, sendMsg, checkMail, ...
- Belief base customisation:
  add, remove, contains, ...
  - Example available with JASON: persistent belief base (in text files, in data bases, ...)

# Further Resources

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



# 2.2 Other BDI-Based Languages

# JADEX

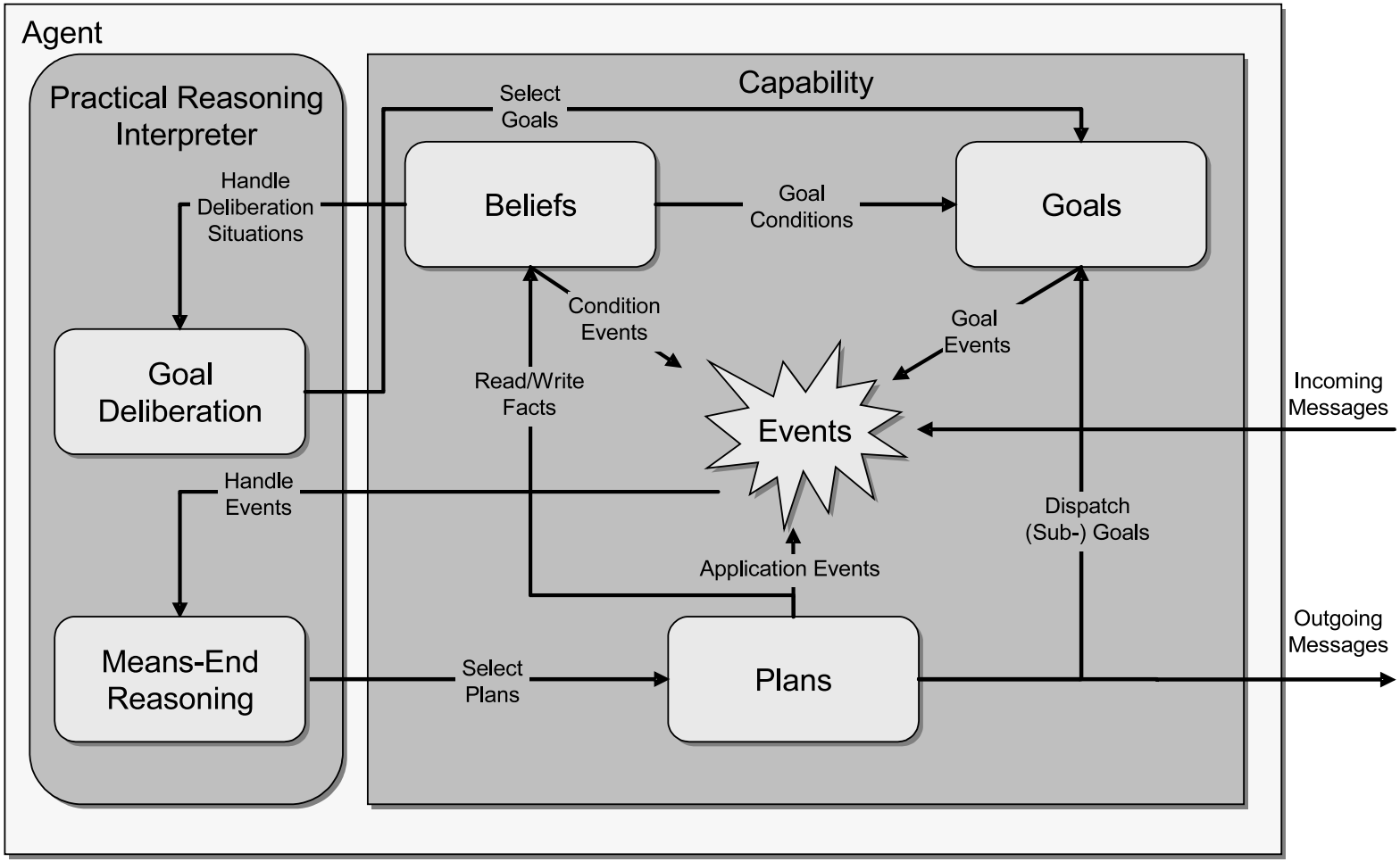



Figure 1 : The Abstract Architecture of JADEX.

# JADEX cont.

- JADEX is a Java-based, modular, and standards compliant, agent platform that allows the development of **goal-oriented agents following the BDI model**.
- It allows for programming intelligent software agents in XML and Java and can be deployed on different kinds of middleware such as JADE.
- `http://jadex-agents.informatik.uni-hamburg.de/`
- [Pokahr et al., 2005, Braubach and Pokahr, 2011].

# 2APL

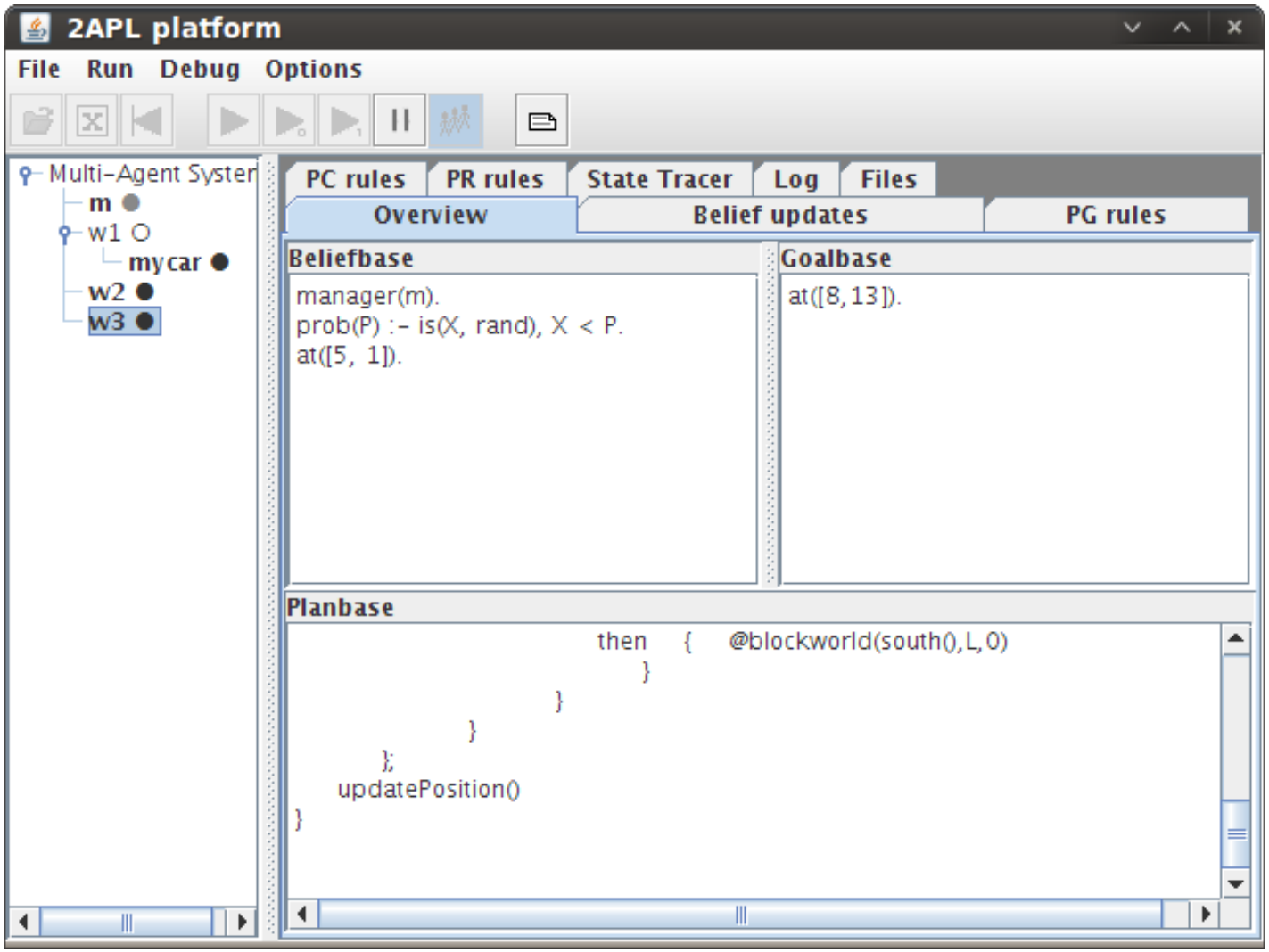



Figure 2 : A Screenshot of the 2APL platform.

# 2APL cont.

- 2APL provides programming constructs both (1) to specify a multiagent system in terms of a set of individual agents and a set of environments, as well as (2) to implement cognitive agents based on the BDI architecture.
- 2APL is a modular programming language allowing the encapsulation of cognitive components in modules. Its graphical interface, through which a user can load, execute, and debug 2APL multiagent programs using different execution modes and several debugging/observation tools.
- `http://apapl.sourceforge.net/`.
- [Dastani, 2008, Alechina et al., 2011].

# AgentFactory

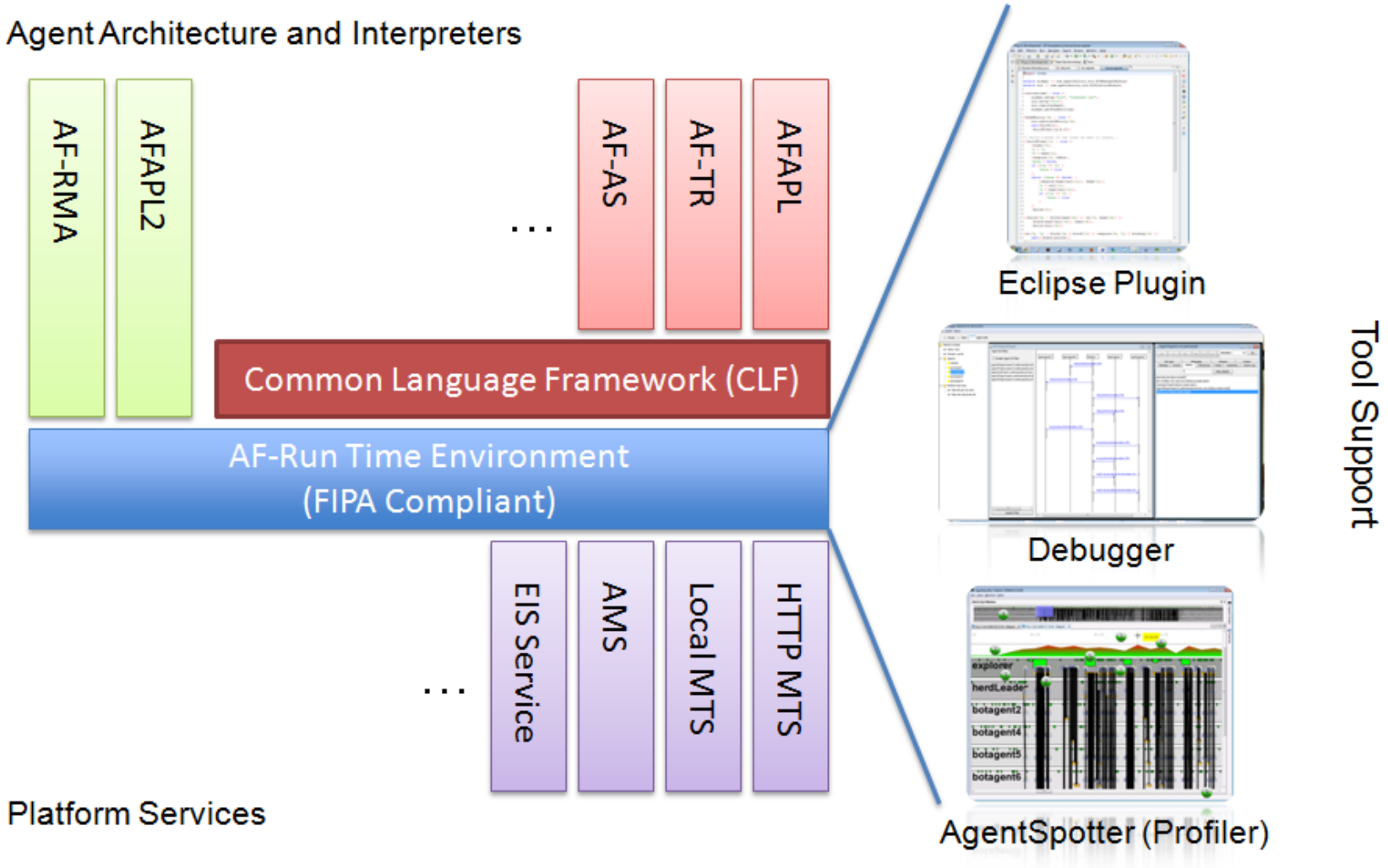
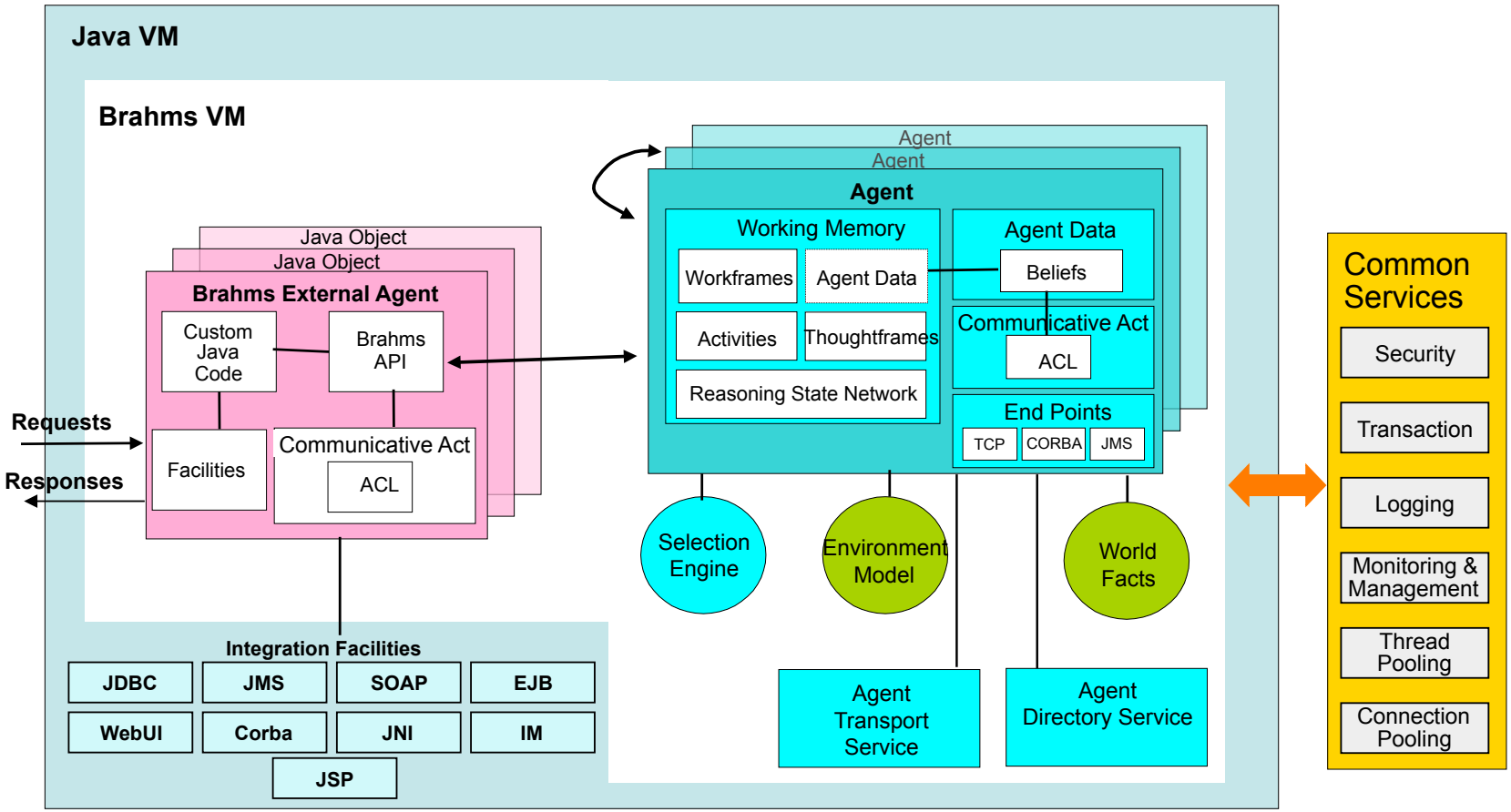



Figure 3 : The Architecture of AgentFactory.

# AgentFactory cont.

- AgentFactory has at its core a FIPA-standards based Run-Time Environment (RTE) that provides support for the deployment of heterogeneous agent types.
- More recent work has resulted in the Common Language Framework, a suite of components for AgentFactory that are intended to help simplify the development of diverse logic-based agent programming languages (APLs).
- `http://www.agentfactory.com`.
- [Lillis et al., 2009, Jordan et al., 2010].

# BRAHMS


Java VM
Brahms VM
Agent
Agent
Agent
Working Memory
Workframes
Agent Data
Activities
Thoughtframes
Reasoning State Network
Agent Data
Beliefs
Communicative Act
ACL
End Points
TCP
CORBA
JMS
Java Object
Java Object
Brahms External Agent
Custom Java Code
Brahms API
Facilities
Communicative Act
ACL
Requests
Responses
Selection Engine
Environment Model
World Facts
Agent Transport Service
Agent Directory Service
Integration Facilities
JDBC
JMS
SOAP
EJB
WebUI
Corba
JNI
IM
JSP
Common Services
Security
Transaction
Logging
Monitoring & Management
Thread Pooling
Connection Pooling

# BRAHMS cont.

- BRAHMS can be seen both as a programming language as well as a behavioural modelling language.
- It allows users to model complex agent organisations, to simulate people, objects and environments.
- A particularly exciting application is the multiagent system OCAMS that was developed with BRAHMS and is running continually in NASA's ISS Mission control: `http://ti.arc.nasa.gov/news/ocams-jsc-award/`.
- `http://ti.arc.nasa.gov/news/ocams-jsc-award/`.
- [Clancey et al., 2003, Stocker et al., 2011, van Putten et al., 2008].

# GOAL

```
init module{
    knowledge{
        clear(table)  . clear(X) :- block(X), not(on(_, X)), not(holding(X)) .
        ...
    }
    % no initial beliefs about block configuration.
    goals{
        on(a,b), on(b,c), on(c,table), on(d,e), on(e,f), on(f,table).
    }
    actionspec{
        pickup(X) { pre{ clear(X), not(holding(_)) }  post{ true } }
        ...
    }
}

% moving X on top of Y is a constructive move if that move results in X being in position.
#define constructiveMove(X, Y) a-goal(tower([X, Y|T])), ... .

main module{
    program{
        if a-goal( holding(X) ) then pickup(X) . % put a block you're holding down.
        if bel( holding(X) ) then {
                if constructiveMove(X,Y) then putdown(X, Y) .
                if true then putdown(X, table) .
        }
    }
}

event module{
    program{
        #define inPosition(X) goal-a( tower([X|T]) ) . % block in position if it achieves a goal.

        % rules for processing percepts (assumes full observability).
        forall bel( block(X), not(percept(block(X))) ) do delete( block(X) ) .
        forall bel( percept(block(X)), not(block(X)) ) do insert( block(X) ) .
        ...
    }
}

module adoptgoal{
        ...
}
```

The **init** module initializes the agent, here by defining knowledge, an initial goal, and action specifications

Macro definitions to create more readable code.

The **main** module is used to code the agent's deliberation using rules for selecting actions.

Rules in the **event** module are used to process percepts and messages that the agent receives.

User-defined modules.

# GOAL cont.

- A GOAL agent program is a set of modules which consist of various sections including knowledge, beliefs, goals, a program section that contains action rules, and action specifications.
- Each of these sections is represented in a knowledge representation language such as Prolog, answer set programming, SQL (or Datalog), or the planning domain definition language.
- `http://mmi.tudelft.nl/trac/goal`.
- [Hindriks and Roberti, 2009, Hindriks, 2007].

# 2.3 Approaches based on executable logics

# Concurrent MetateM

- MetateM is a programming language for multiagent systems based on a first-order temporal logic (with discrete, linear models with finite past and infinite future) [Fisher, 1997].
- Concurrent MetateM is the concurrent extension of MetateM [Fisher, 1996].
- A Concurrent MetateM system contains a number of concurrently executing agents which are able to communicate through message passing.
- Each agent executes a first-order temporal logic specification of its desired behaviour.

# Concurrent METATEM cont.

- An agent has two main components: (1) an **interface** which defines how the agent may interact with its environment (i.e. other agents), (2) a **computational engine**, defining how the agent may act.
- The computational engine of an agent is based on the METATEM paradigm of *executable temporal logics*.
- The idea behind this approach is to directly execute a declarative agent specification given as a set of **program rules** which are temporal logic formulae of the form: *"antecedent about past $\rightarrow$ consequent about future"*. The intuitive interpretation of such a rule is **"on the basis of the past, do so in the future"**.

# Con-Golog, Indi-Golog

- ConGolog ([Giacomo et al., 2000]) and IndiGolog ([Giacomo et al., 2009]) are languages extending Golog, a language based on the situation calculus introduced by McCarthy. Golog stands for alGOl in LOGic.
- Actions are described as in the classical STRIPS approach: they have **preconditions** that must be satisfied in order to apply the action. The **postcondition** then describes the change of the world.
- The evolution of the world is described within the logical language by **fluents**, which are terms in the language. The effects of an action is formalised by successor-state axioms: they describe what the successor state of a given state looks like if an action is applied.

# Con-Golog, Indi-Golog cont.

- Golog is a programming language that hides the application of the situation calculus and is thus much more user-friendly.
- Procedures in Golog actions are reduced to primitive actions which refer to **actions in the real world**, such as picking up objects, opening doors, moving from one room to another, and so on.

# 3. Organisation and Environment Programming

3 Organisation and Environment Programming
- MOISE
- CARTAGO

# Organisations and Environments

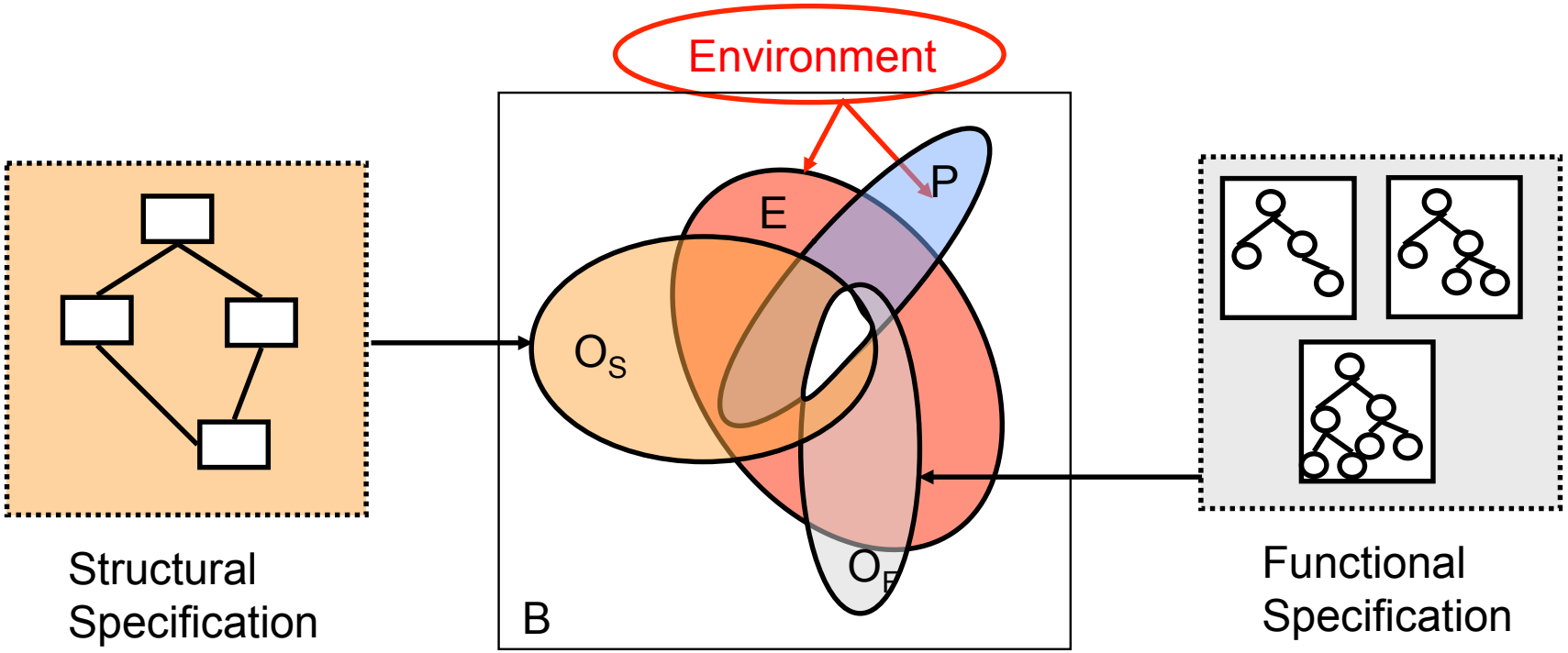
- There are many approaches to agent organisations and agent environments
- Not many are practical enough to use in multi-agent systems development
- In these slides we will look particularly at *M*OISE for programming organisations and CARTAGO for programming environments

# 3.1 MOISE

# MOISE Framework

- MOISE Organisation Modelling Language as Tag-based language (issued from MOISE [Hannoun et al., 2000], MOISE$^+$ [Hübner et al., 2002a], MOISEINST [Gâteau et al., 2005])
- OMI developped as an artefact-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])
  - dedicated *organisational artefacts* that provide general services for the agents to work within an organisation
  - *organisational agents* that monitor and manage the functioning of the organisation
- Dedicated integration bridges for
  - Agents and Environment (c4Jason, c4Jadex [Ricci et al., 2009a])
  - Environment and Organisation ( [Piunti et al., 2009]

# MOISE+ Modelling Dimensions




Groups, links, roles
*Compatibilities, multiplicities*
*inheritance*

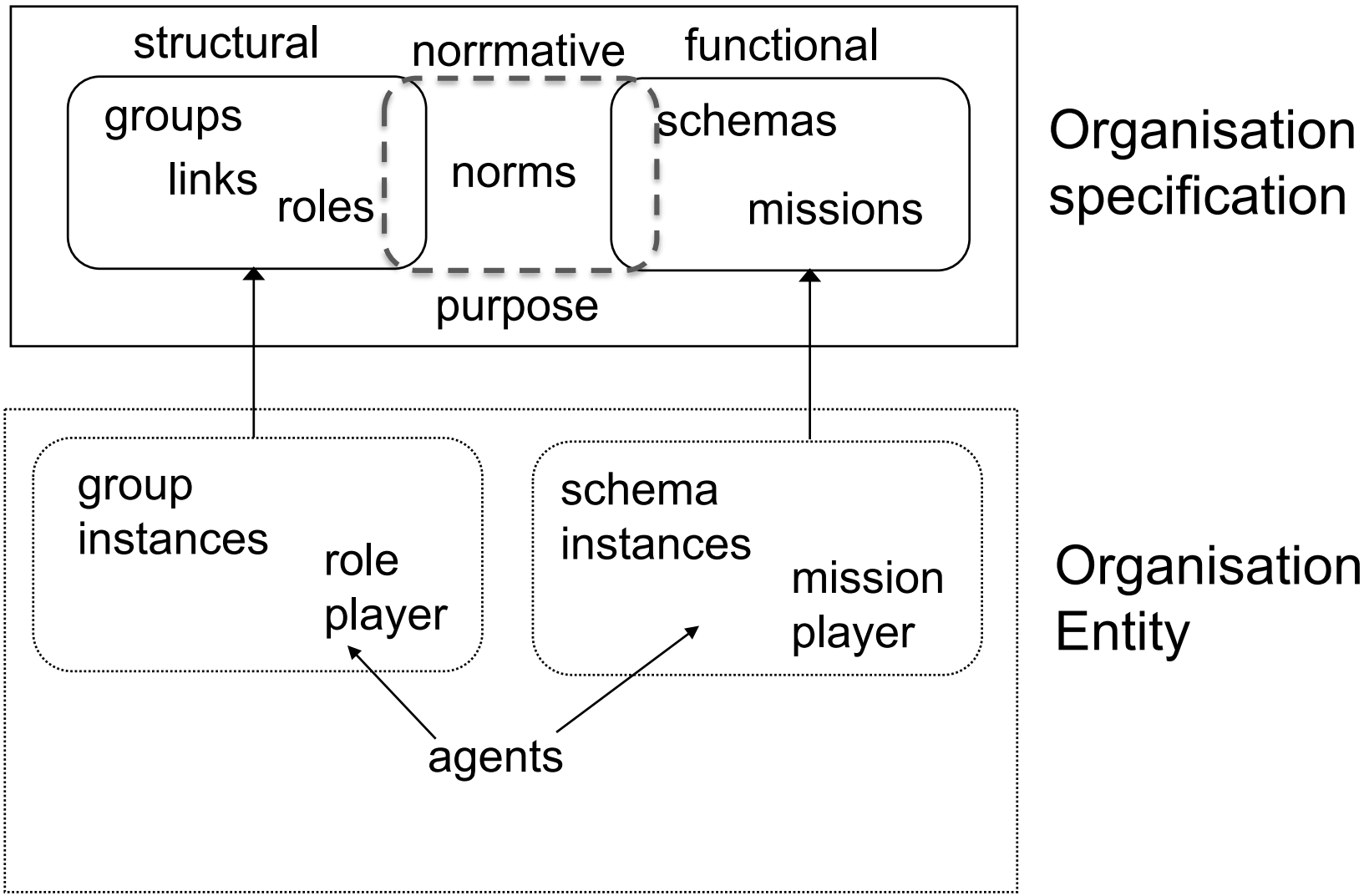
Normative Specification
Permissions, Obligations
Allows agents autonomy!

Global goals, plans,
Missions, schemas,
preferences

# MOISE+ OML

- OML for defining organisation specification and organisation entity
- Three independent dimensions [Hübner et al., 2007] ($\rightsquigarrow$ 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
    - $\rightsquigarrow$ $\mathcal{J}$-MOISE+ [Hübner et al., 2007]
  - the Organisation Management Infrastructure
    - $\rightsquigarrow$ ORA4MAS [Hübner et al., 2009]

# MOISE OML global picture



# 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 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

# MOISE OML Structural Specification I

- Defined with the tag structural-specification in the context of an organisational-specification
- One section for definition of all the roles participating to the structure of the organisation (role-definitions tag)
- Specification of the group including all sub-group specifications (groupe-specification tag)

# MOISE OML Structural Specification II

## Example 3.1

# MOISE OML Structural Specification III

```
<organisational-specification
  <structural-specification>
     <role-definitions>  ... </role-definitions>
     <group-specification id="xxx">
       ...
     </group-specification>
  </structural-specification>
  ...
</organisational-specification>
```

# Role Specification I

- Role definition(role tag) in role-definitions section, is composed of:
  - identifier of the role (id attribute of role tag)
  - inherited roles (extends tag) - by default, all roles inherit of the soc role -

# Role Specification II

## Example 3.2

# Role Specification III

```
<role-definitions>
  <role id="player" />
  <role id="coach" />
  <role id="middle"> <extends role="player"/> </role>
  <role id="leader"> <extends role="player"/> </role>
  <role id="r1>
    <extends role="r2" />
    <extends role="r3" />
  </role>
  ...
</role-definitions>
```

# Group Specification I

- Group definition (group-specification tag) is composed of:
  - group identifier (id attribute of group-specification tag)
  - roles participating to this group and their cardinality (roles tag and id, min, max), i.e. min. and max. number of agents that should adopt the role in the group (default is 0 and unlimited)
  - links between roles of the group (link tag)
  - subgroups and their cardinality (sub-groups tag)
  - formation constraints on the components of the group (formation-constraints)

# Group Specification II

## Example 3.3

```
<group-specification id="team">
   <roles>
        <role id="coach" min="1" max="2"/> ...
   </roles>
   <links> ... </links>
   <sub-groups> ... </sub-groups>
   <formation-constraints> ... </formation-constraints>
</group-specification>
```

# Link Specification I

- Link definition (link tag) included in the group definition is composed of:
  - role identifiers (from, to)
  - type (type) with one of the following values: authority, communication, acquaintance
  - scope of the link (scope): inter-group, intra-group
  - validity in sub-groups: if extends-sub-group set to true, the link is also valid in all sub-groups (default false)

# Link Specification II

## Example 3.4

```
<link from="coach"
      to="player"
      type="authority"
      scope="inter-group"
      extends-sub-groups="true" />
```

# Formation Constraint Specification I

- Formation constraints definition (formation-constraints tag) in a group definition is composed of:
  - compatiblity constraints (compatibility tag) between roles (from, to), with a scope, extends-sub-groups and directions (bi-dir)

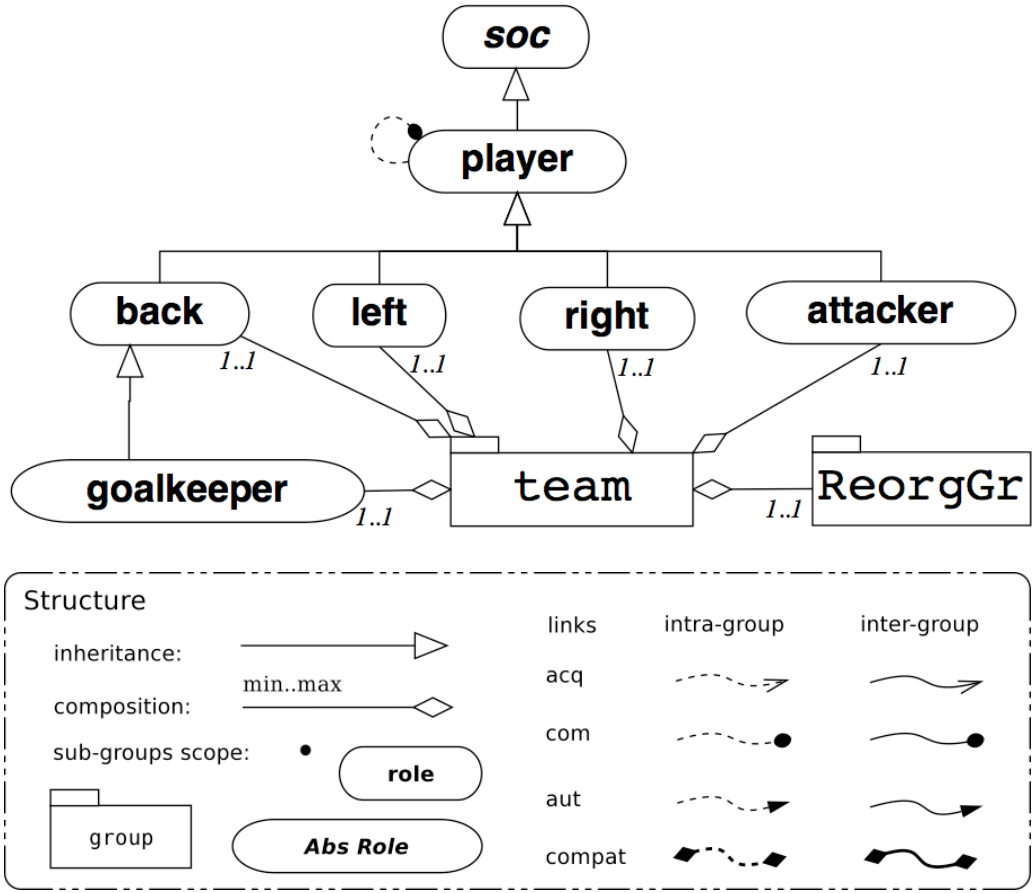
# Formation Constraint Specification II

## Example 3.5

# Formation Constraint Specification III

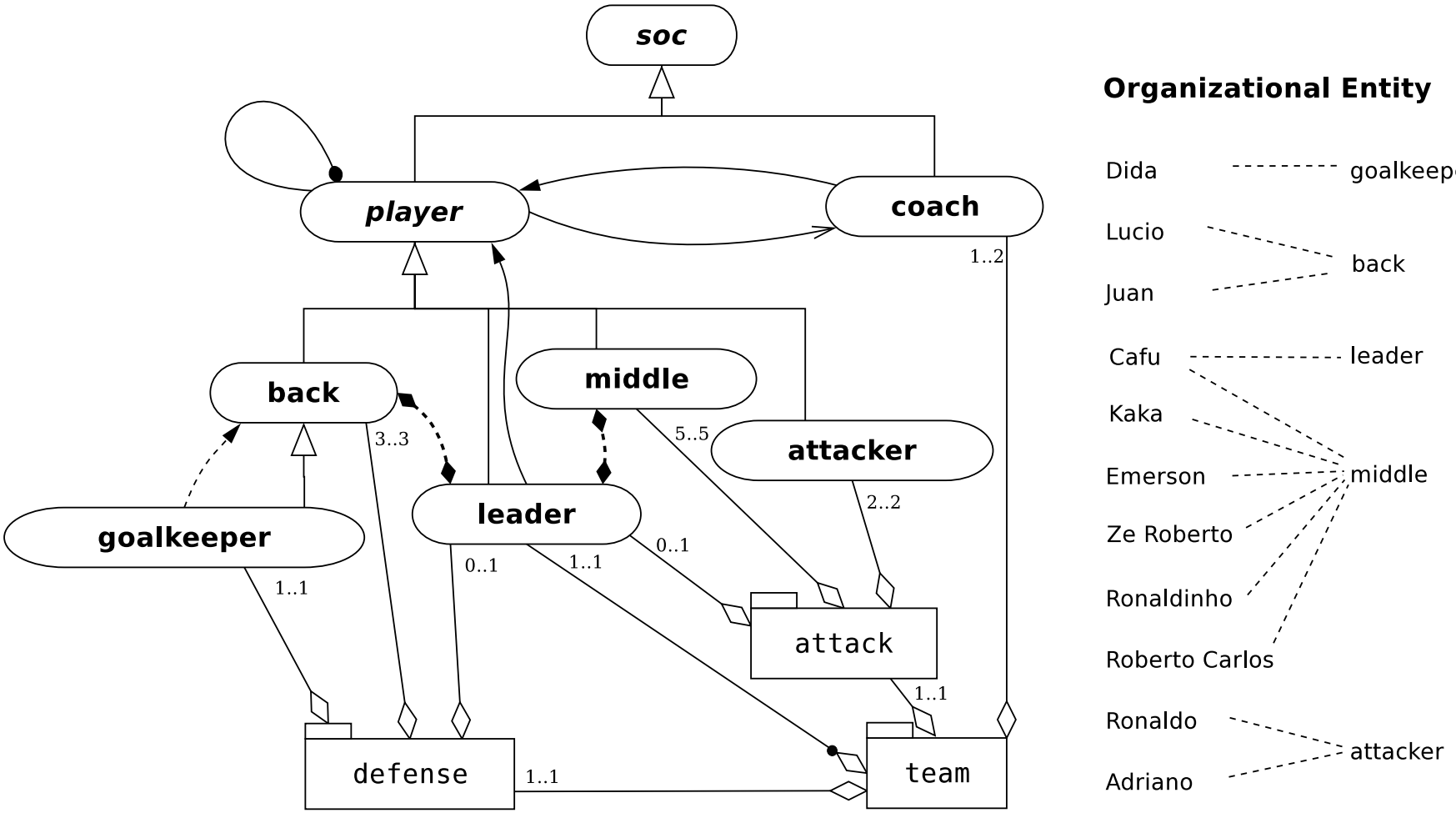
```
<formation-constraints>
  <compatibility from="middle"
                 to="leader"
                 scope="intra-group"
                 extends-sub-groups="false"
                 bi-dir="true"/>
  ...
</formation-constraints>
```

# Structural Specification Example I




Graphical representation of structural specification of Joj Team

# Structural Specification Example II




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

# MOISE OML Functional Specification I

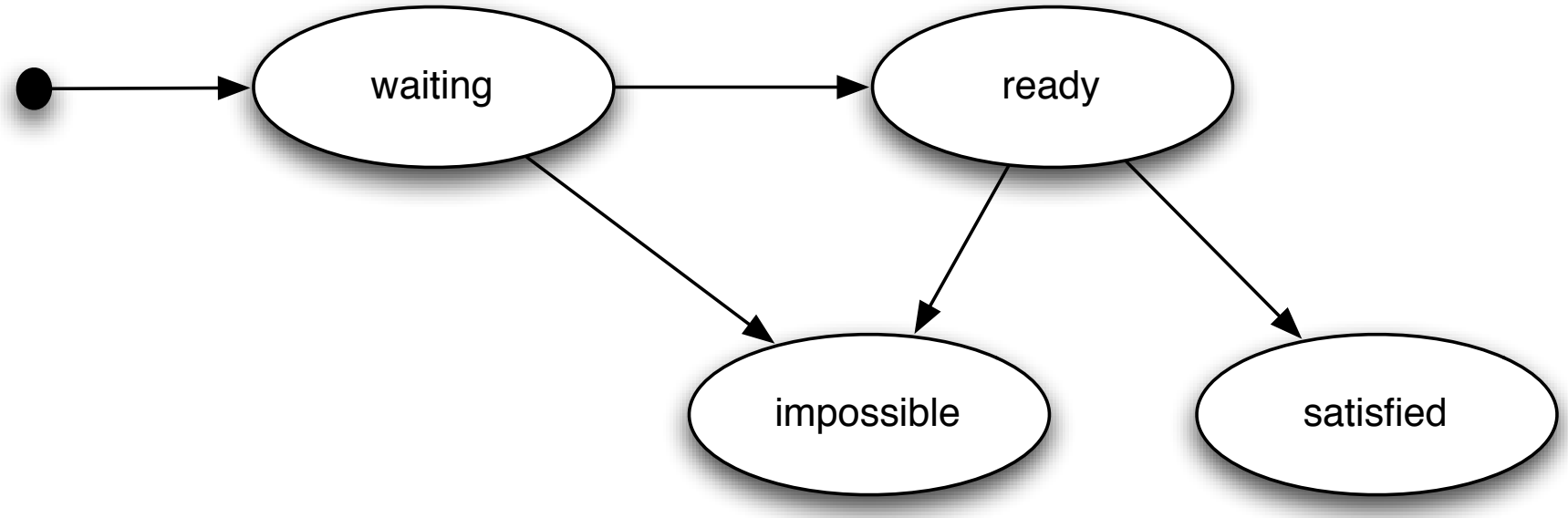
- Specifies the expected behaviour of an MAS in terms of *goals* along two levels:
  - *Collective* with *Scheme*
  - *Individual* with *Mission*
- Components:
  - *Goals*:
    - *Achievement goal* (default type). Goals of this type should be declared as satisfied by the agents committed to them, when achieved

# MOISE OML Functional Specification II

- *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

# Goal States




| | |
|---|---|
| waiting | initial state |
| ready | 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 Functional Specification I

- Defined with the tag functional-specification in the context of an organisational-specification
- Specification in sequence of the different schemes participating to the expected behaviour of the organisation

# MOISE OML Functional Specification II

## Example 3.6

# MOISE OML Functional Specification III

```
<functional-specification>
    <scheme id="sideAttack" >
        <goal id="dogoal" > ... </goal>
        <mission id="m1" min="1" max="5">
            ...
        </mission>
        ...
    </scheme>
    ...
</functional-specification>
```

# Scheme Specification I

- Scheme definition (scheme tag) is composed of:
  - identifier of the scheme (id attribute of scheme tag)
  - the root goal of the scheme with the plan aiming at achieving it (goal tag)
  - the set of missions structuring the scheme (mission tag)
- Goal definition within a scheme (goal tag) is composed of:
  - an idenfier (id attribute of goal tag)
  - a type (achievement default or maintenance)

# Scheme Specification II

- min. number of agents that must satisfy it (min) (default is “all”)
- optionally, an argument (argument tag) that must be assigned to a value when the scheme is created
- optionally a plan

- Plan definition attached to a goal (plan tag) is composed of
  - one and only one operator (operator attribute of plan tag) with sequence, choice, parallel as possible values
  - set of goal definitions (goal tag ) concerned by the operator

# Scheme Specification Example

```
<scheme id="sideAttack">
 <goal id="scoreGoal" min="1" >
  <plan operator="sequence">
    <goal id="g1" min="1" ds="get the ball" />
    <goal id="g2" min="3" ds="to be well placed">
      <plan operator="parallel">
        <goal id="g7" min="1" ds="go toward the opponent's field" />
        <goal id="g8" min="1" ds="be placed in the middle field" />
        <goal id="g9" min="1" ds="be placed in the opponent's goal area" />
      </plan>
    </goal>
    <goal id="g3" min="1" ds="kick the ball to the m2Ag" >
       <argument id="M2Ag" />
    </goal>
    <goal id="g4" min="1" ds="go to the opponent's back line" />
    <goal id="g5" min="1" ds="kick the ball to the goal area" />
    <goal id="g6" min="1" ds="shot at the opponent's goal" />
  </plan>
 </goal>
 ...
```

# Mission Specification I

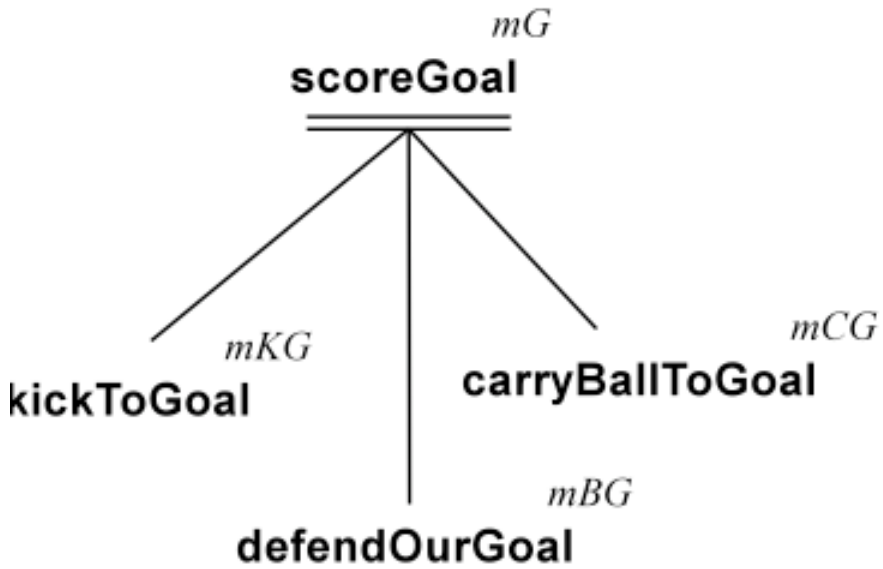
- Mission definition (mission tag) in the context of a scheme definition, is composed of:
  - identifier of the mission (id attribute of mission tag)
  - cardinality of the mission min (0 is default), max (unlimited is default) specifying the number of agents that can be committed to the mission
  - the set of goal identifiers (goal tag) that belong to the mission

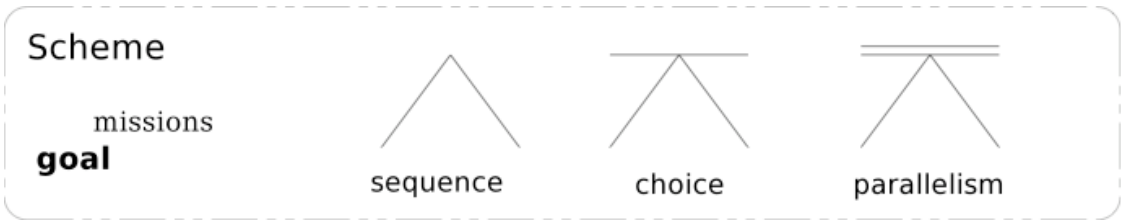
# Mission Specification II

## Example 3.7

```
<scheme id="sideAttack">
  ... the goals ...
 <mission id="m1" min="1" max="1">
    <goal id="scoreGoal" /> <goal id="g1" />
    <goal id="g3" /> ...
  </mission>
  ...
</scheme>
```

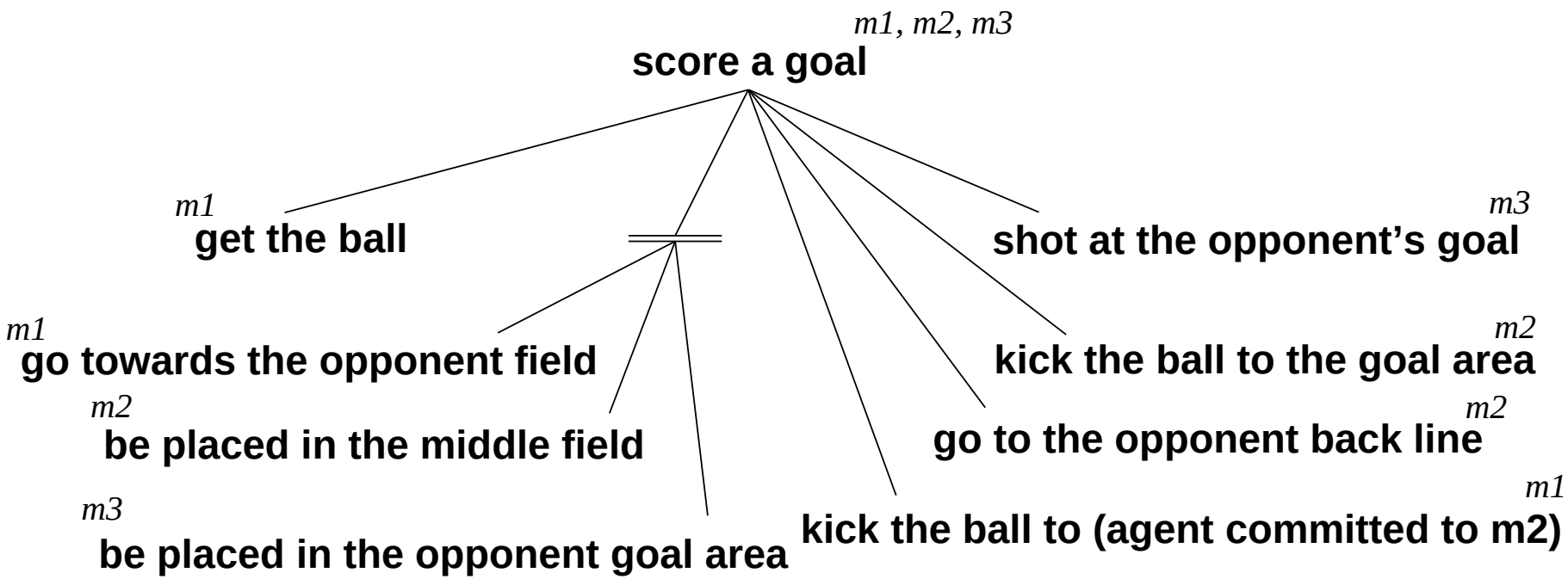
# Functional Specification Example (1)



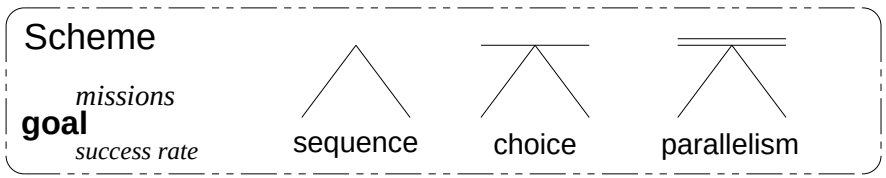




Graphical representation of social scheme for joj team

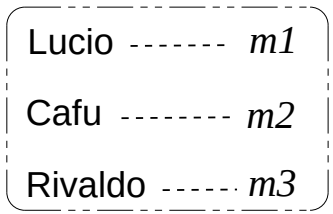
# Functional Specification Example (2)




Key




Organizational Entity




Graphical representation of social scheme "side_attack" for joj team

# 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

# MOISE OML Normative Specification

- Defined with the tag normative-specification in the context of an organisational-specification
- Specification in sequence of the different norms participating to the governance of the organisation

## Example 3.8

```
<normative-specification>
    <norm id="n1" ... />
     ...
    <norm id="..." ... />
</normative-specification>
```

# Norm Specification

- Norm definition (norm tag) in the context of a normative-specification definition, is composed of:
  - the identifier of the norm (id)
  - the type of the norm (type) with obligation, permission as possible values
  - optionally a condition of activation (condition) with the following possible expressions:
    - checking of properties of the organisation (e.g. #role_compatibility, #mission_cardinality, #role_cardinality, #goal_non_compliance)
    - ⇝ unregimentation of organisation properties!!!
    - (un)fulfillment of an obligation stated in a particular norm (unfulfilled, fulfilled)
  - the identifier of the role (role) on which the role is applied
  - the identifier of the mission (mission) concerned by the norm
  - optionally a time constraint (time constraint)

# Norm Specification – example

| role | deontic | mission | | TTF |
|---|---|---|---|---|
| *back* | *obliged* | $m1$ | get the ball, go ... | 1 minute |
| *left* | *obliged* | $m2$ | be placed at ..., kick ... | 3 minute |
| *right* | *obliged* | $m2$ | | 1 day |
| *attacker* | *obliged* | $m3$ | kick to the goal, ... | 30 seconds |

```
<norm id = "n1" type="obligation"
      role="back" mission="m1" time-constraint="1 minute"/>
...
<norm id = "n4" type="obligation"
      condition="unfulfilled(obligation(_,n2,_,_))"
      role="coach" mission="ms" time-constraint="3 hour"/>
...
```

# Organisation Entity Dynamics

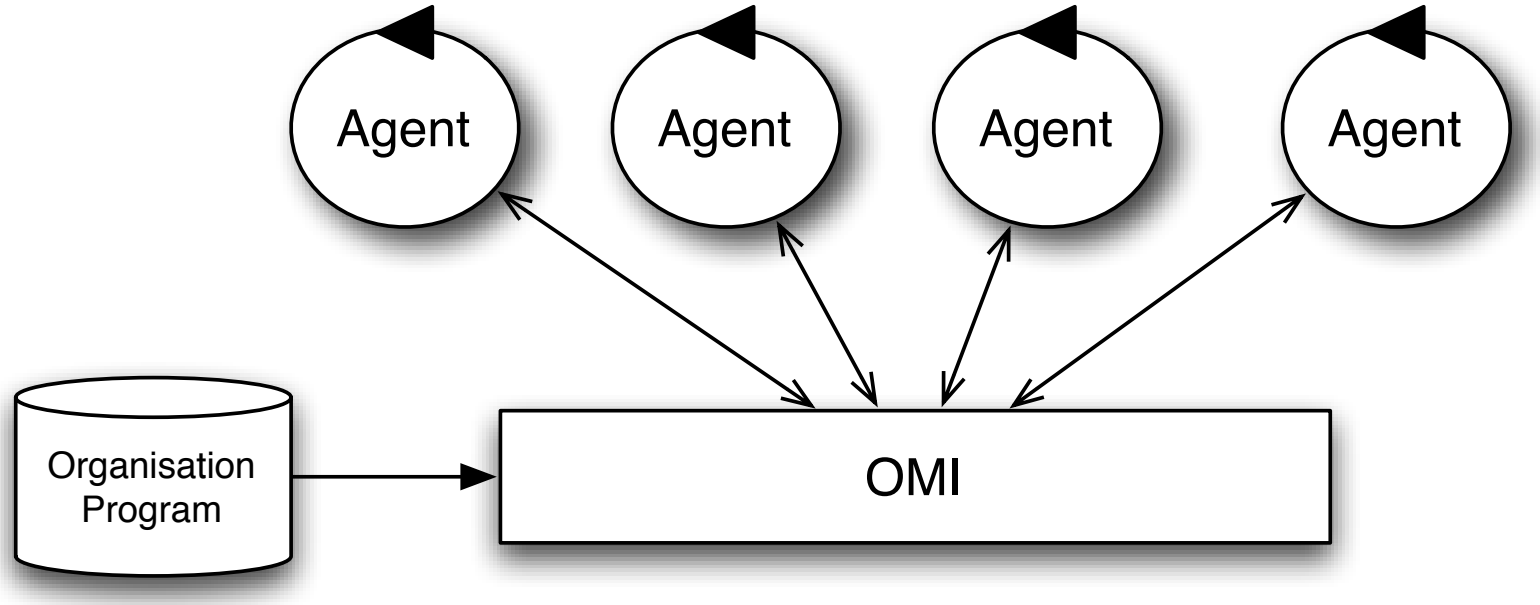
1. Organisation is created (by the agents)
   - instances of groups
   - instances of schemes
2. Agents enter into groups *adopting* roles
3. Groups become *responsible* for schemes
   - Agents from the group are then obliged to commit to missions in the scheme
4. Agents *commit* to missions
5. Agents *fulfil* mission's goals
6. Agents leave schemes and groups
7. Schemes and groups instances are destroyed

# Organisation management infrastructure (OMI) I

## Responsibility

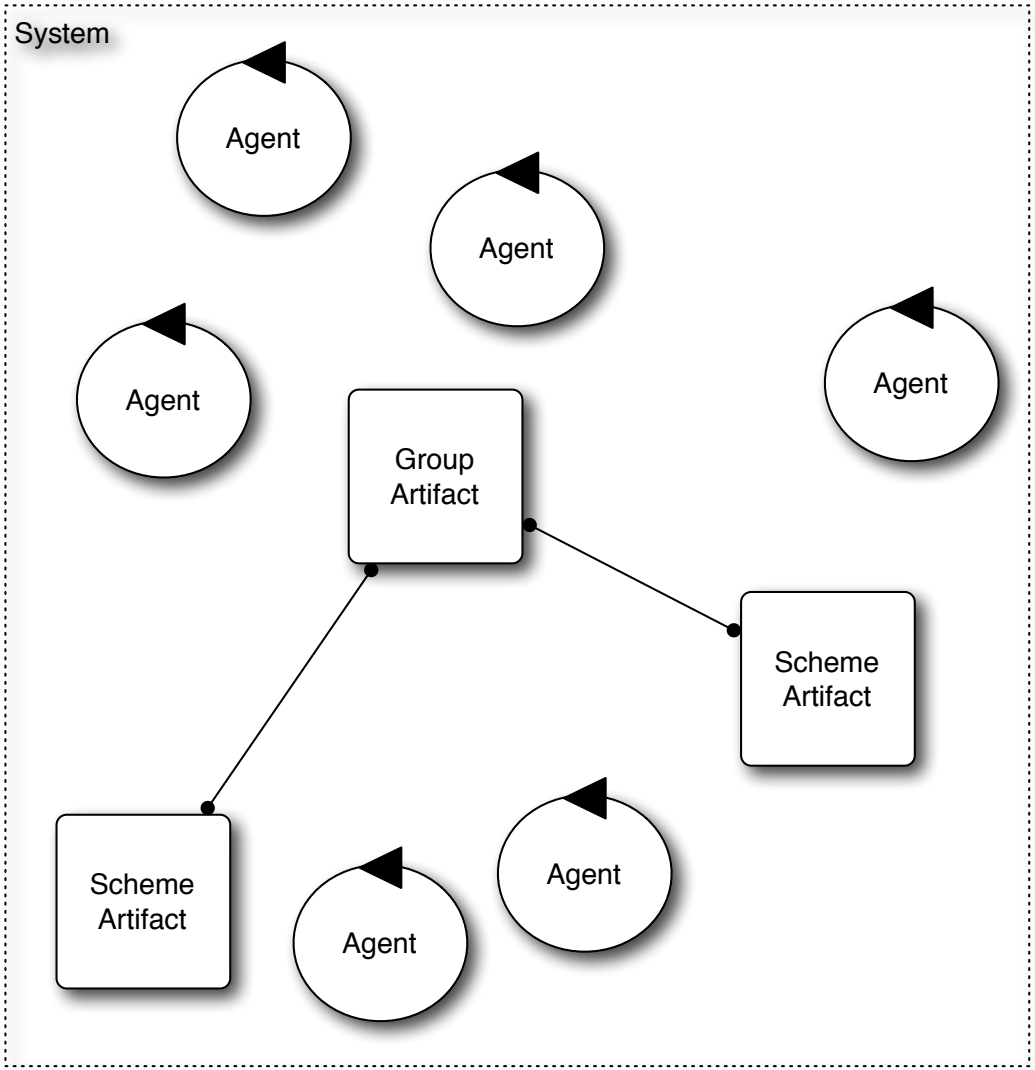
- Managing – coordination, regulation – the agents' execution within organisation defined in an organisational specification

# Organisation management infrastructure (OMI) II



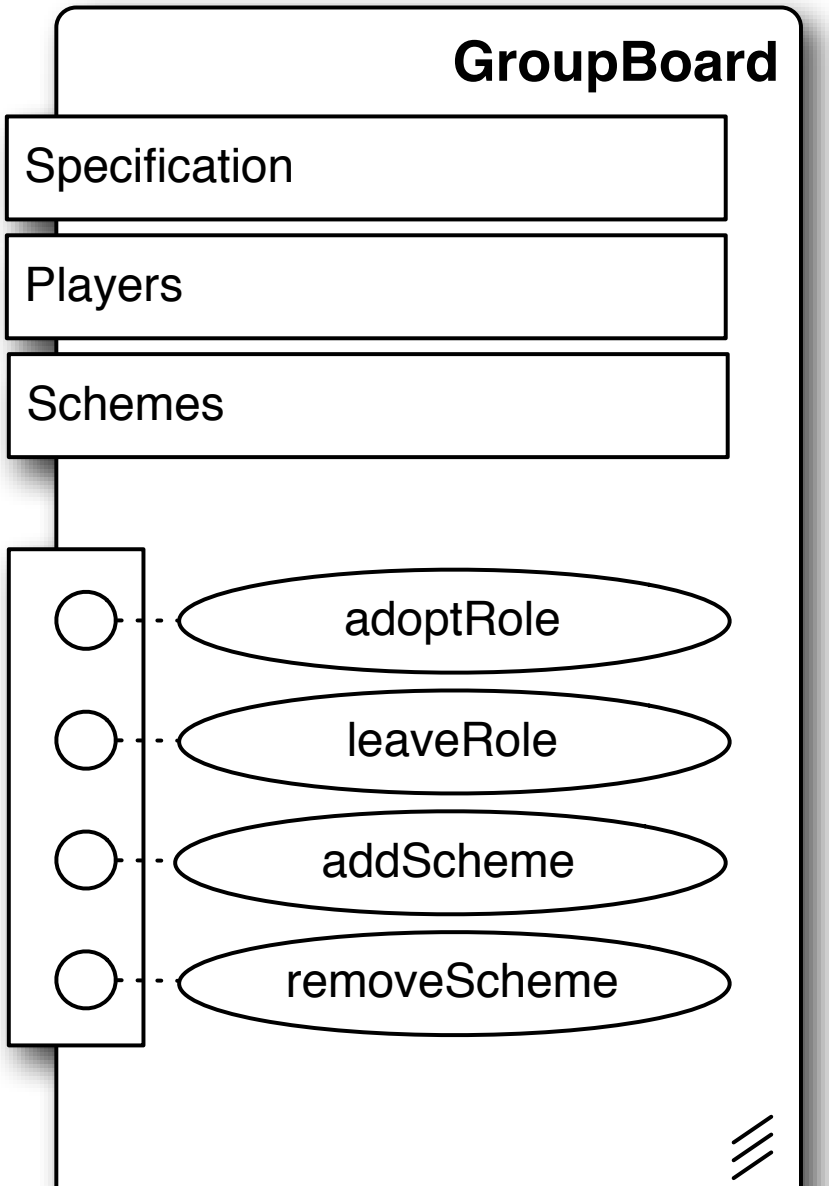

(e.g. MadKit, AMELI, $\mathcal{S}$-MOISE$^+$, ...)

# Organisational artifacts in ORA4MAS




- based on A&A and MOISE
- agents create and handle organisational artifacts
- artifacts in charge of *regimentations*, detection and evaluation of norms compliance
- agents are in charge of decisions about sanctions
- distributed solution

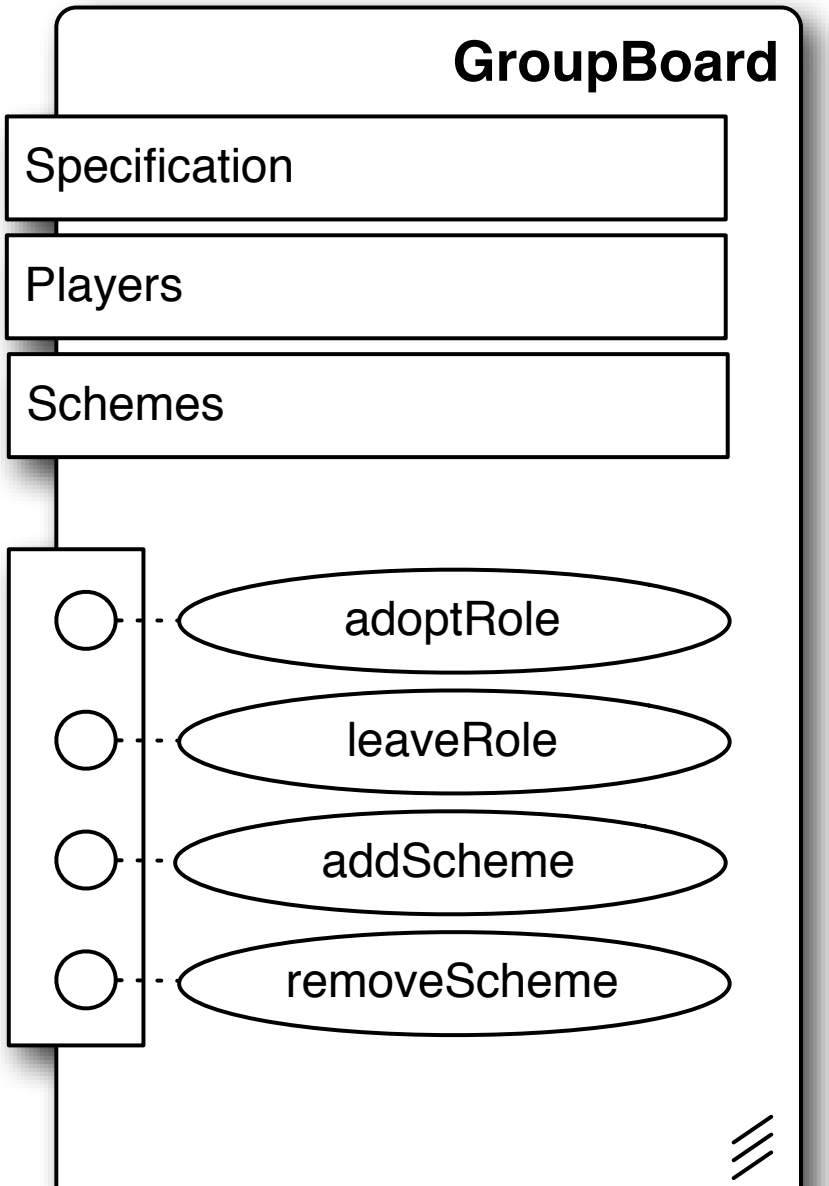
# ORA4MAS – GroupBoard artifact




*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 – GroupBoard artifact




*Observable Properties*:

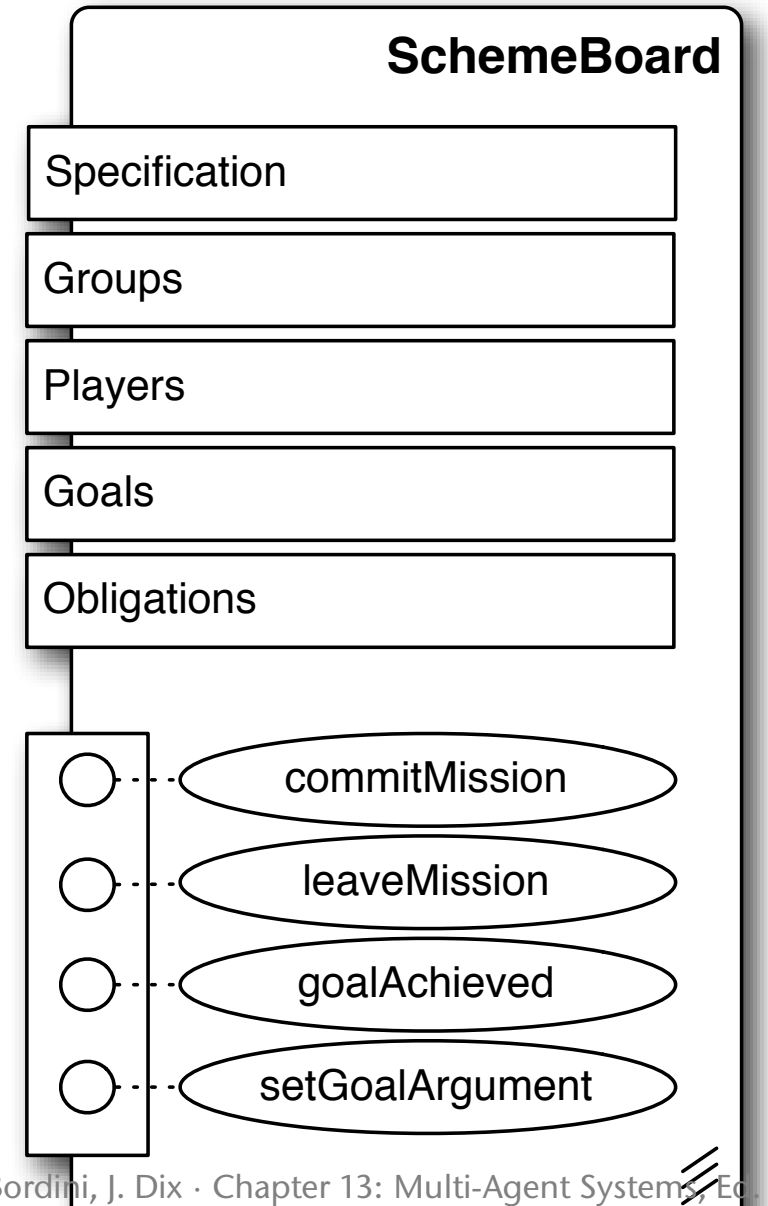
- specification: the specification of the group in the OS (an object of class moise.os.ss.Group)
- players: a list of agents playing roles in the group. Each element of the list is a pair (agent x role)
- schemes: a list of scheme identifiers that the group is responsible for

# 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

# 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

# Organisational Artifact Implementation I

- Organisational artifacts are programmed with a Normative Programming Language (NPL) [Hübner et al., 2010]
- The NPL *norms* have
  - an activation condition
  - a consequence
- two kinds of consequences are considered
  - regimentations
  - obligations

# Organisational Artifact Implementation II

## Example 3.9 (norm)

# Organisational Artifact Implementation III

```
    norm n1: plays(A,writer,G) -> fail.
```

or

```
    norm n1: plays(A,writer,G)
          -> obligation(A,n1,plays(A,editor,G),
             'now + 3 min').
```

# Agent integration

- Agents can interact with organisational artifacts as with ordinary artifacts by perception and action

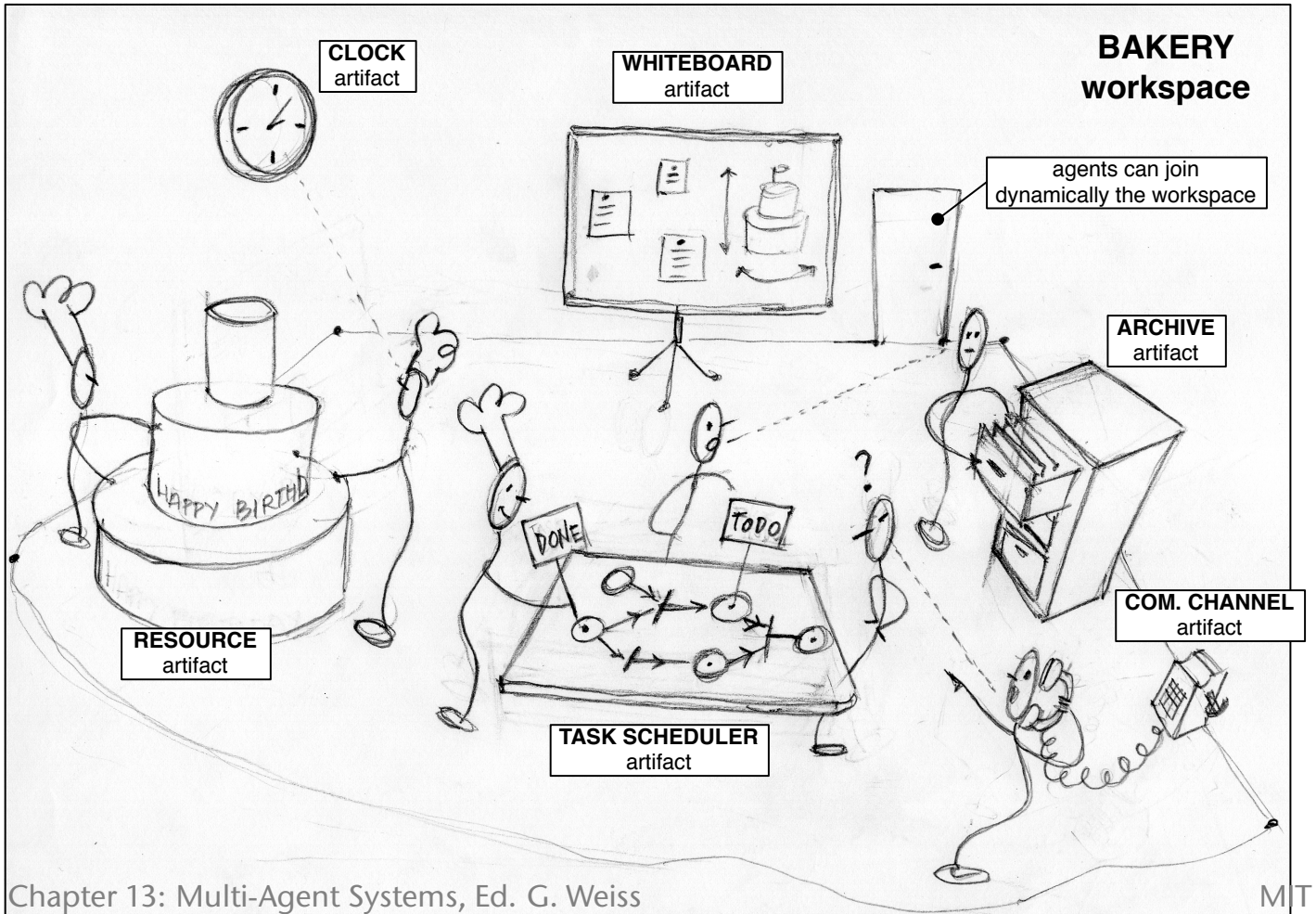
⇝ Any Agent Programming Language integrated with CARTAGO can use organisational artifacts

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*

# 3.2 CArtAgO

# Agents and Artifacts (A&A) Conceptual Model: Background Human Metaphor



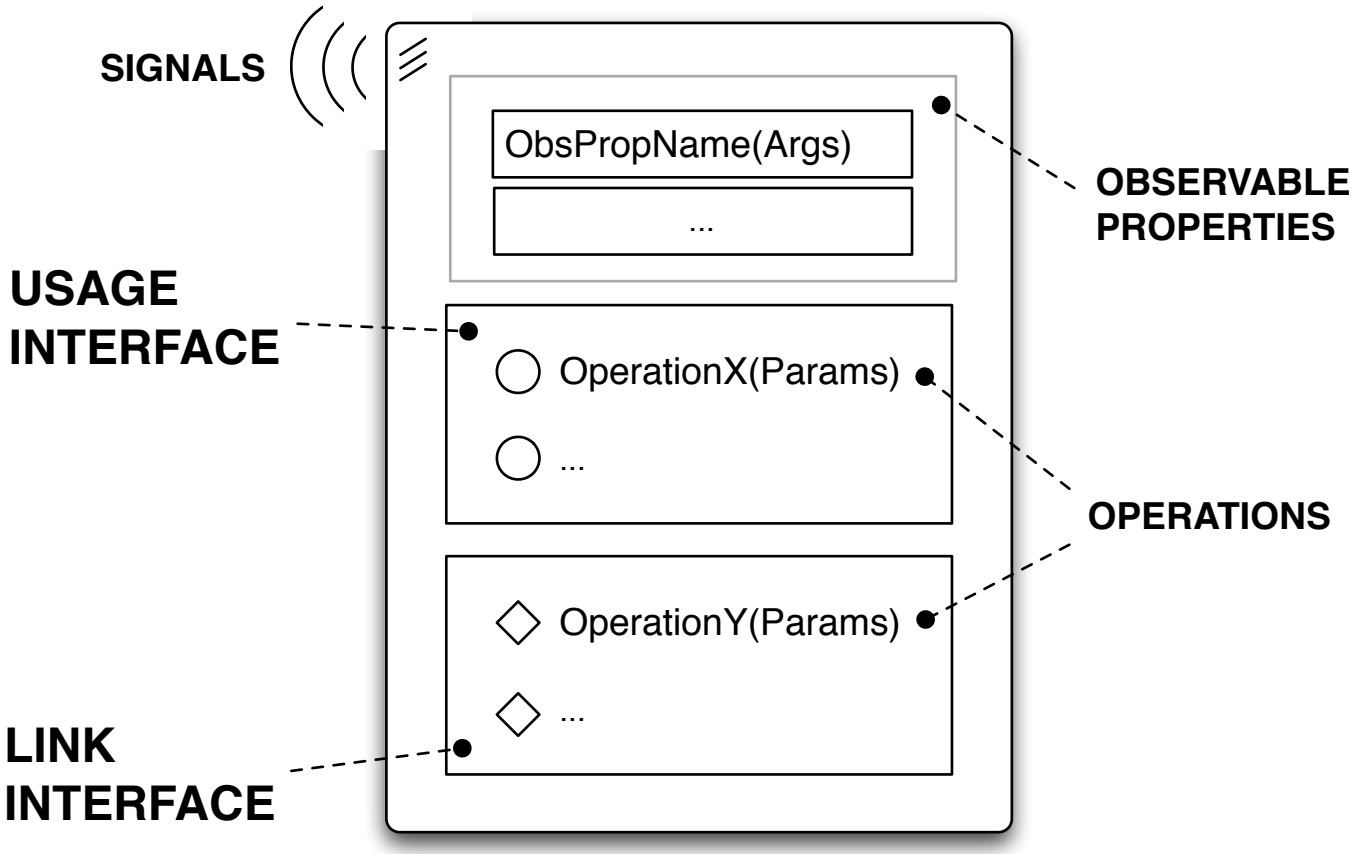
# A&A Basic Concepts [Omicini et al., 2008]

- 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

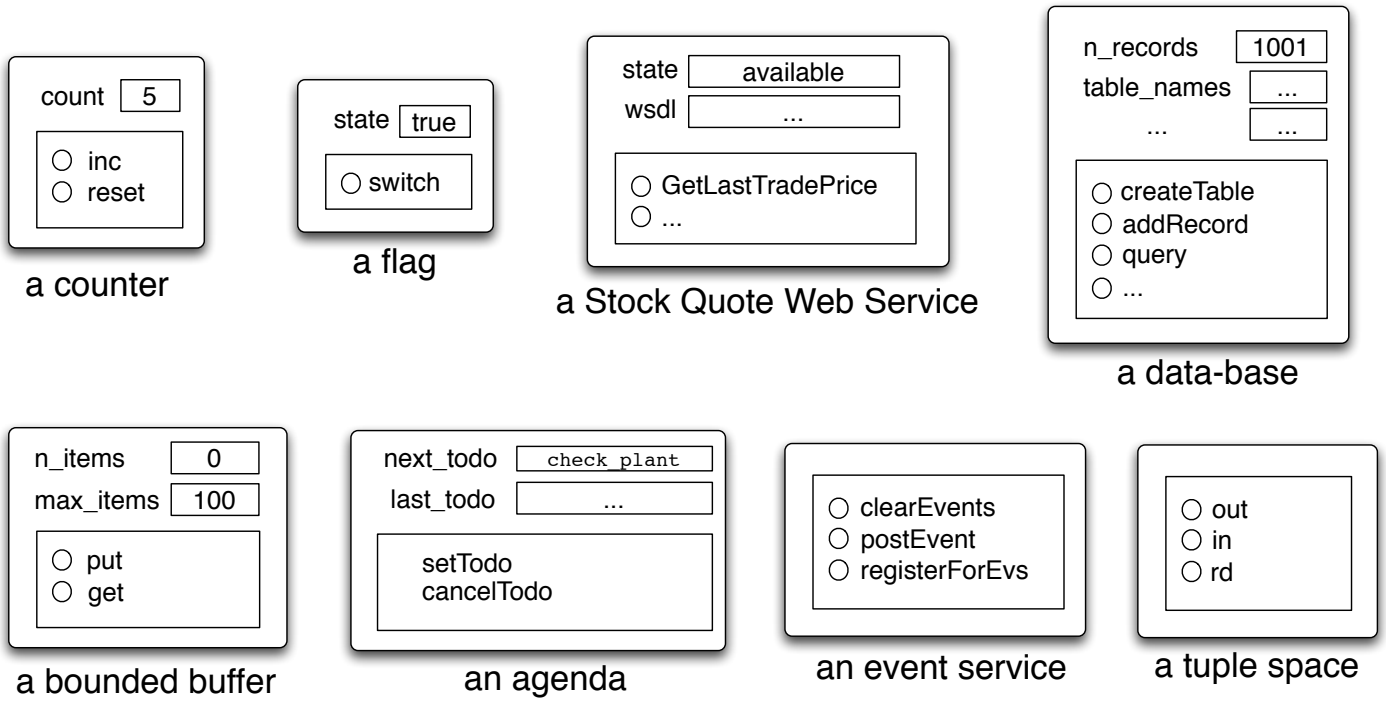
# A&A Programming Model Features [Ricci et al., 2007a]

- 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

# Artifact Abstract Representation



# A World of Artifacts



# 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 I

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

# Actions and Percepts in Artifact-Based Environments II

## actions $\longleftrightarrow$ artifacts' operation

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

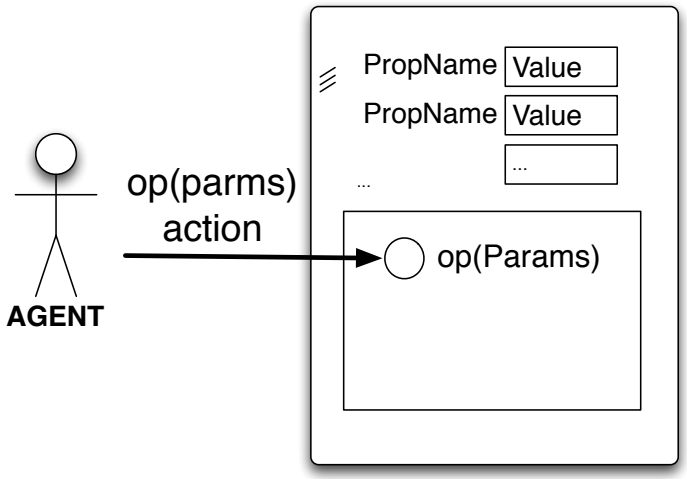
- action success/failure semantics is defined by operation semantics

# Actions and Percepts in Artifact-Based Environments III

percepts $\longleftrightarrow$ artifacts' observable properties + signals

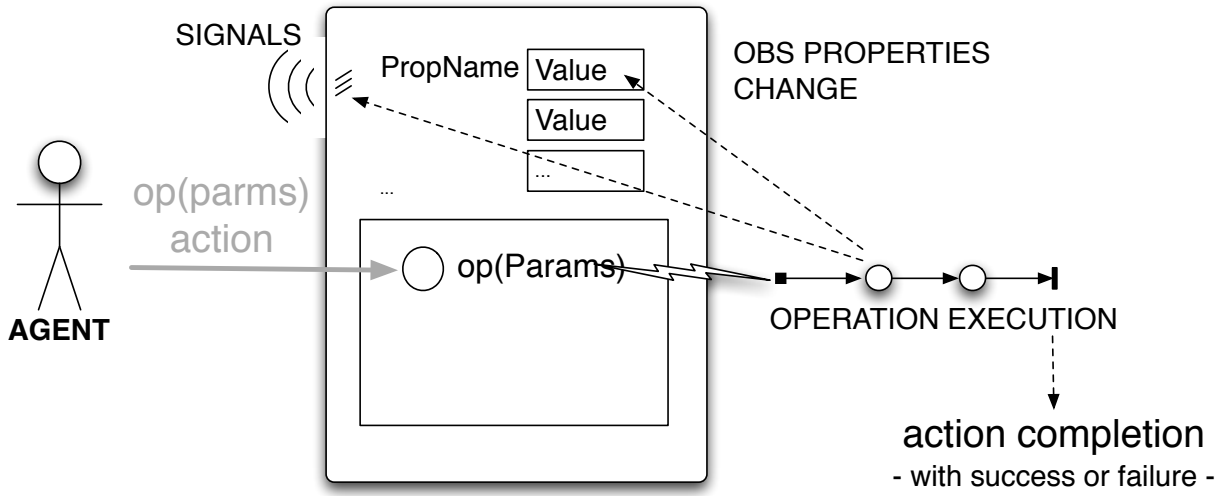
properties represent percepts about the state of the environment signals represent percepts concerning events signalled by the environment

# Interaction Model: Use



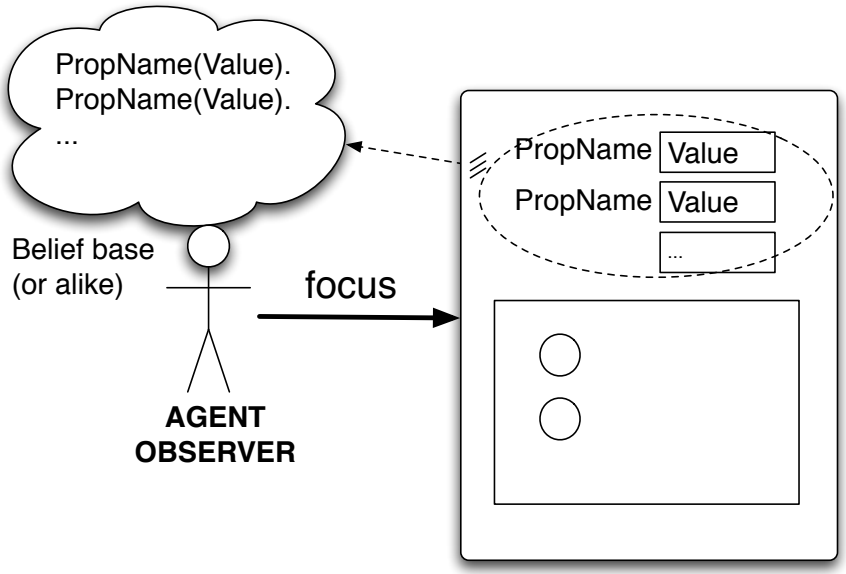

- Performing an action corresponds to triggering the execution of an operation
  - acting on artifact?s usage interface

# Interaction Model: Operation execution



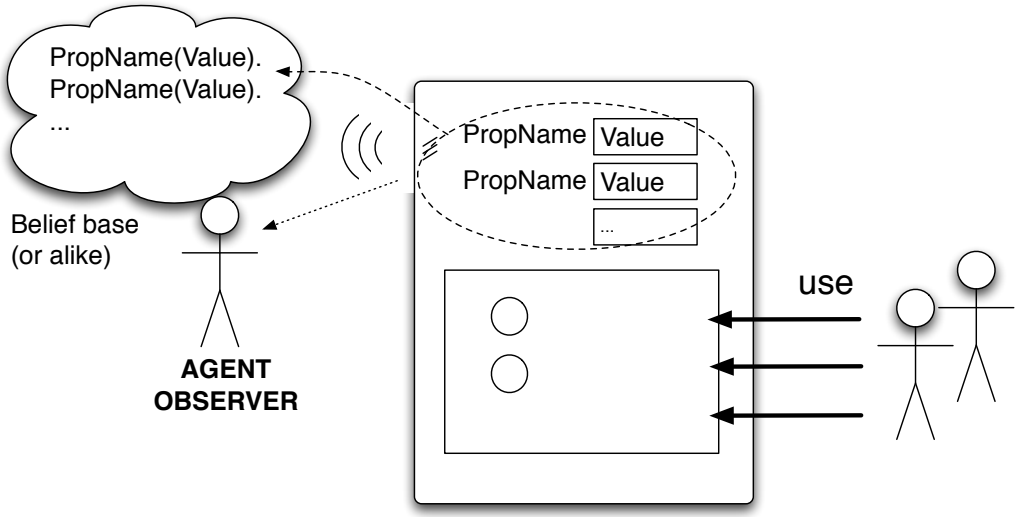

- a process structured in one or multiple transactional steps
- asynchronous with respect to agent
  - *...which can proceed possibly reacting to percepts and executing actions of other plans/activities*
- operation completion causes action completion
  - action completion events with success or failure

# Interaction Model: Observation



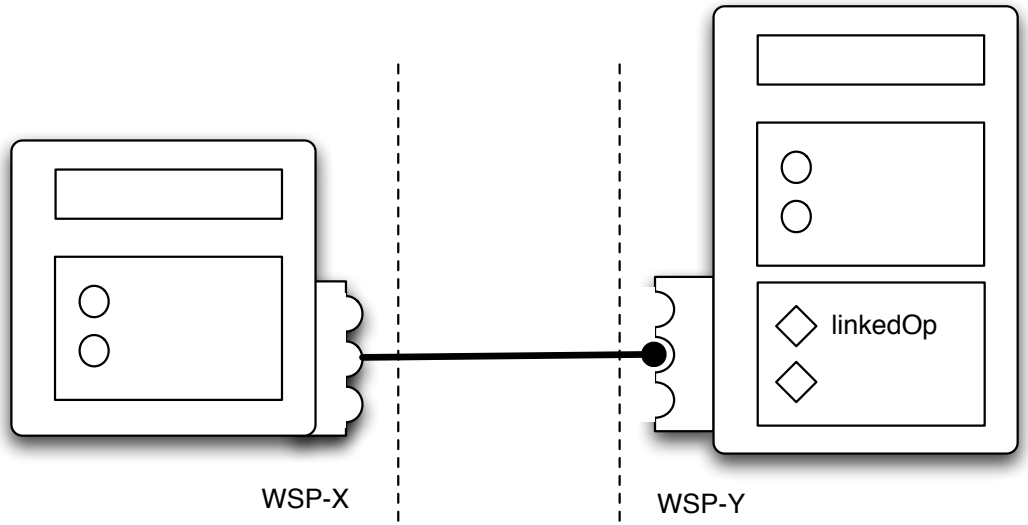

- Agents can dynamically select which artifacts to observe
  - predefined `focus`/`stopFocus` actions

# Interaction Model: Observation




- 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

# Artifact Linkability




- 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., 2009b]
- Computational framework / infrastructure to implement and run artifact-based environment [Ricci et al., 2007b]
  - 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 `http://cartago.sourceforge.net`

# Integration with Agent Languages and Platforms

- Integration with existing agent platforms [Ricci et al., 2008]
  - available bridges: JASON, Jadex, AgentFactory, simpA, ...
    - ongoing: ***2APL***
  - including organisation platforms: *M*OISE framework [Hübner et al., 2002b, Hübner et al., 2006]
- 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

# 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

# 4. An Example in JaCaMo

4 An Example in JaCaMo
- Organisation Program
- Agent Programs
- Environment Program

# Introduction

- Running example used in Chapters 13 to 15
- Scenario introduced in more details in Chapter 15
- The design was made with the focus of demonstrating the JaCaMo approach rather than the best solution for the problem
- We show here code excerpts from the 3 JaCaMo levels: agent, organisation, and envirnoment
- The full running example can be downloaded from `http://www.inf.pucrs.br/r.bordini/WeissBookChapter13Ex`

# Scenario

- The chosen running example centers mostly on the organisation level and, in this design, partly on the environment
- Agents are very simple and in general they only execute the required action at the required time in orchestration with the team, which is mostly handled by the organisation
- Left as exercise to extend to multiple units; there are both Jason and CArtAgO solutions for contract net
- The assembly cell of a manufacturing plant is assumed to have two jigs in a rotating table, with two manufacturing robots located at two ends of the table: one that mostly does loading and unloading tasks and another that is able to join separate parts that have been loaded into a jig

# Summary of the Manufacturing Process (quoted from Chapter 15) I

1. robot1 loads an A part into one of the jigs on the rotating table
2. robot1 loads a B part on top of it
3. the table rotates so the A and B parts are at robot2
4. robot2 joins the parts together, yieldig an “AB” part
5. the table rotates back to robot1
6. robot1 moves the AB part to the flipper

# Summary of the Manufacturing Process (quoted from Chapter 15) II

7. the flipper flips the part over (“BA”) at the same time as robot1 loads a C part into the jig
8. the BA part is loaded on top of the C part
9. the table rotates
10. robot2 joins the C and BA parts, yielding a complete ABC part
11. the table is rotated, and
12. robot1 then unloads the finished part.

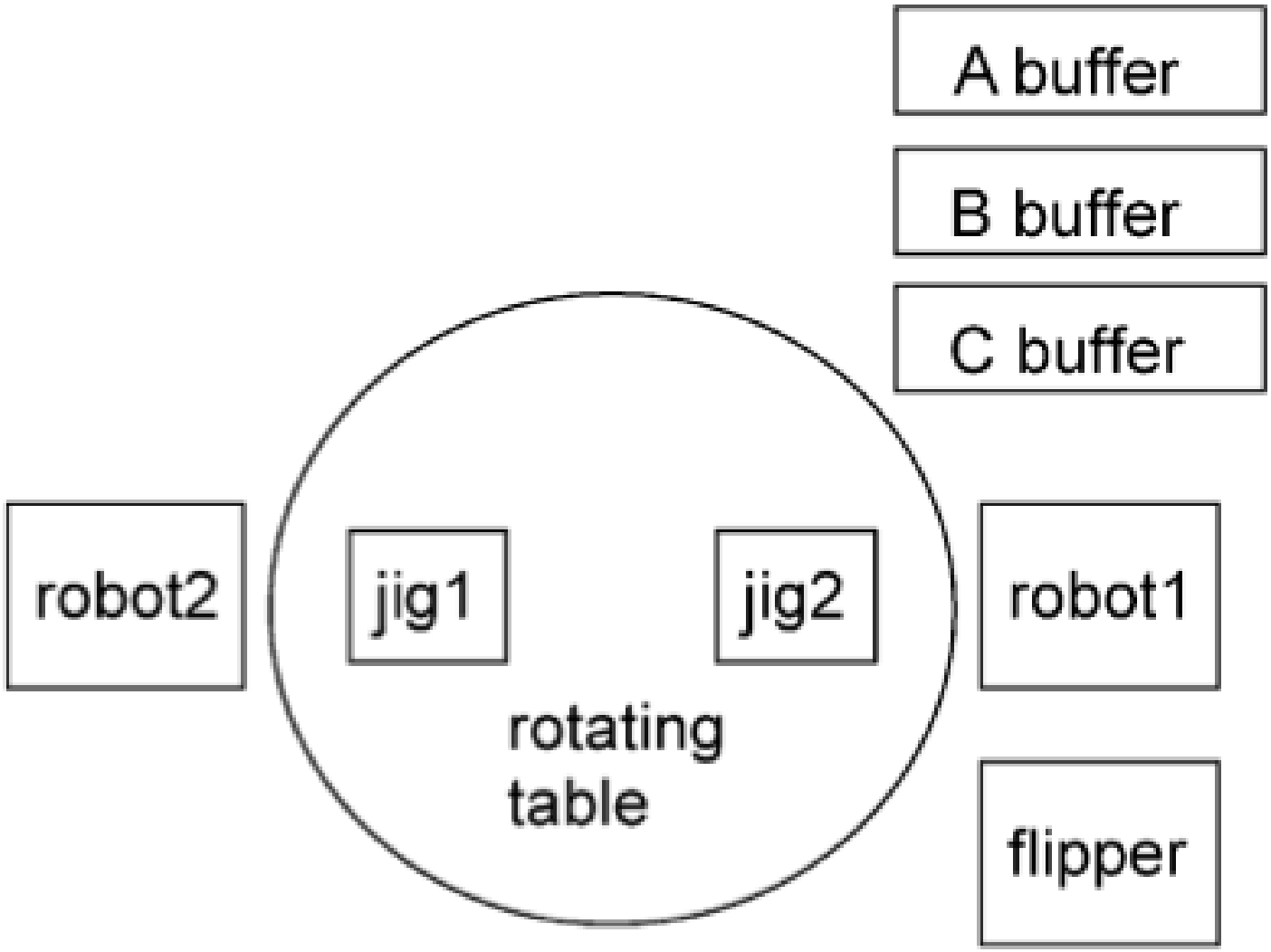
# Summary of the Manufacturing Process (quoted from Chapter 15) III

- Although this process may sound straightforward, it is made more complex by the need to manage a number of concurrent assembly jobs. In other words, we want to be able to exploit parallelism, for instance having robot2 be assembling one part while robot1 is unloading a different order. On the other hand, we need to respect synchronization requirements such as not moving the table while robot1 or robot2 are operating.

# Summary of the Manufacturing Process (quoted from Chapter 15) IV

- Note that in general in *holonic manufacturing* there are multiple interchangeable entities so that the process of selecting a table, or an assembly robot, needs some mechanism to manage load-balancing (e.g. using contract net).

# Overview of a Manufacturing Cell (from Chapter 15)



# 4.1 Organisation Program

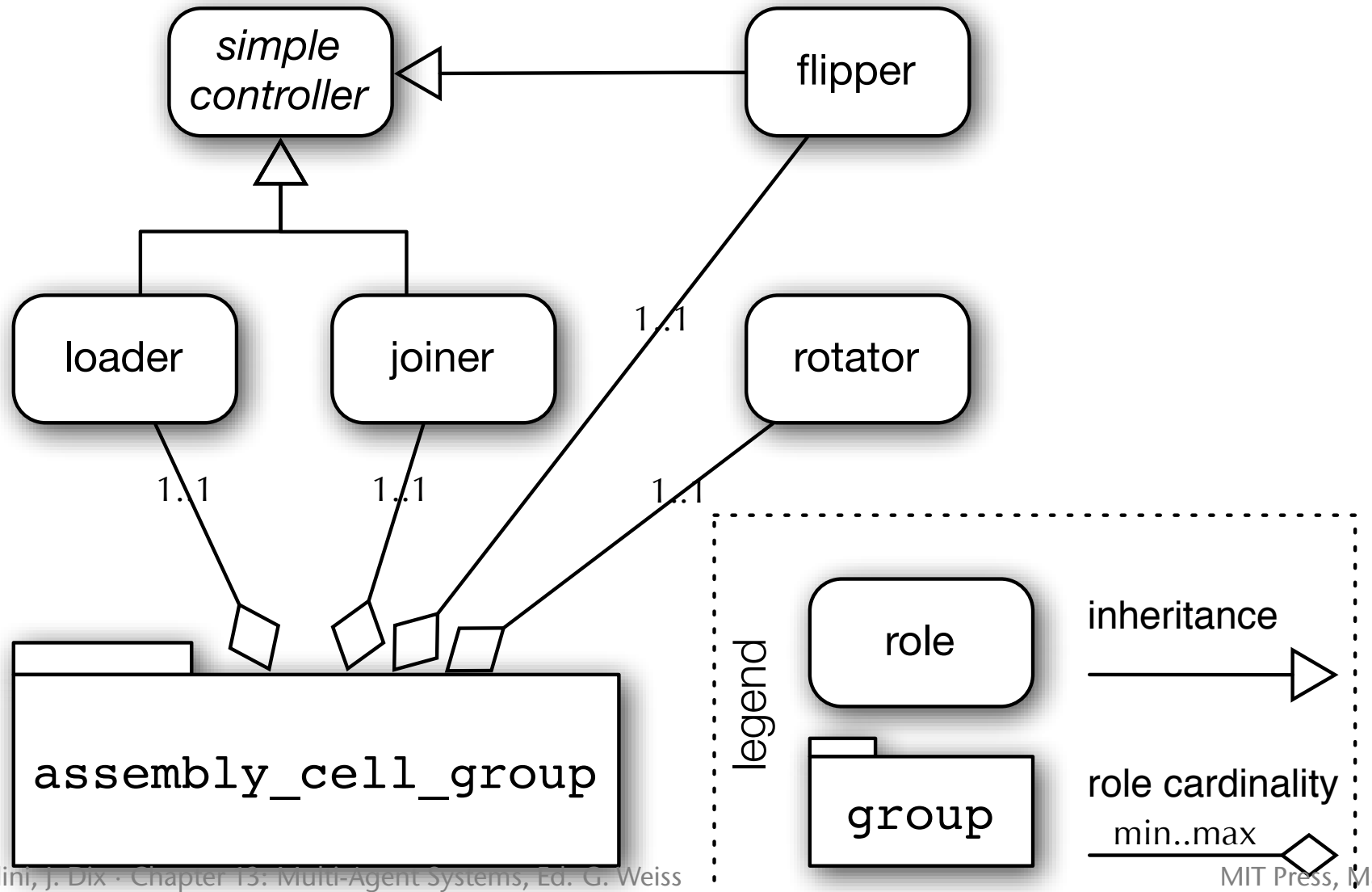
# Organisation I

- Very simple to specify this coordination problem in *M*OISE
- Agent roles:
  - "loader" robot
  - "joiner" robot
  - flipper (controller of the flipping mechanism)
  - rotator (controller of the rotating mechanism)
- the first 3 roles above inherit from role "simple controller" and they all use the same JASON code

# Organisation II

- Agents playing each of the 4 roles neeed for a functional “assembly cell group”
- This example only uses some of the *M*OISE expression power
- Agent “cellmngr” used only to simulate the allocation of assembling tasks, for testing; it uses an artefact where manufacturing requests appear
- an instance of the `manufacture_ABC` scheme (see functional specification) is created for each accepted task: as there are two jigs a cell can concurrently manufacture two pieces

# MOISE Organisation: Structural Specification



# The Social Plan I

- The Functional Specification defines the whole social plan
- For readability, we have divided the whole task into three main parts:
  1. assembling an AB part, then
  2. the complete ABC part, and finally
  3. finishing up (by unloading the complete piece).
- In this manufacturing process most tasks are sequential; the only two tasks that are to be done in parallel are indicated by the parallel horizontal lines in that diagram (see also the legend)

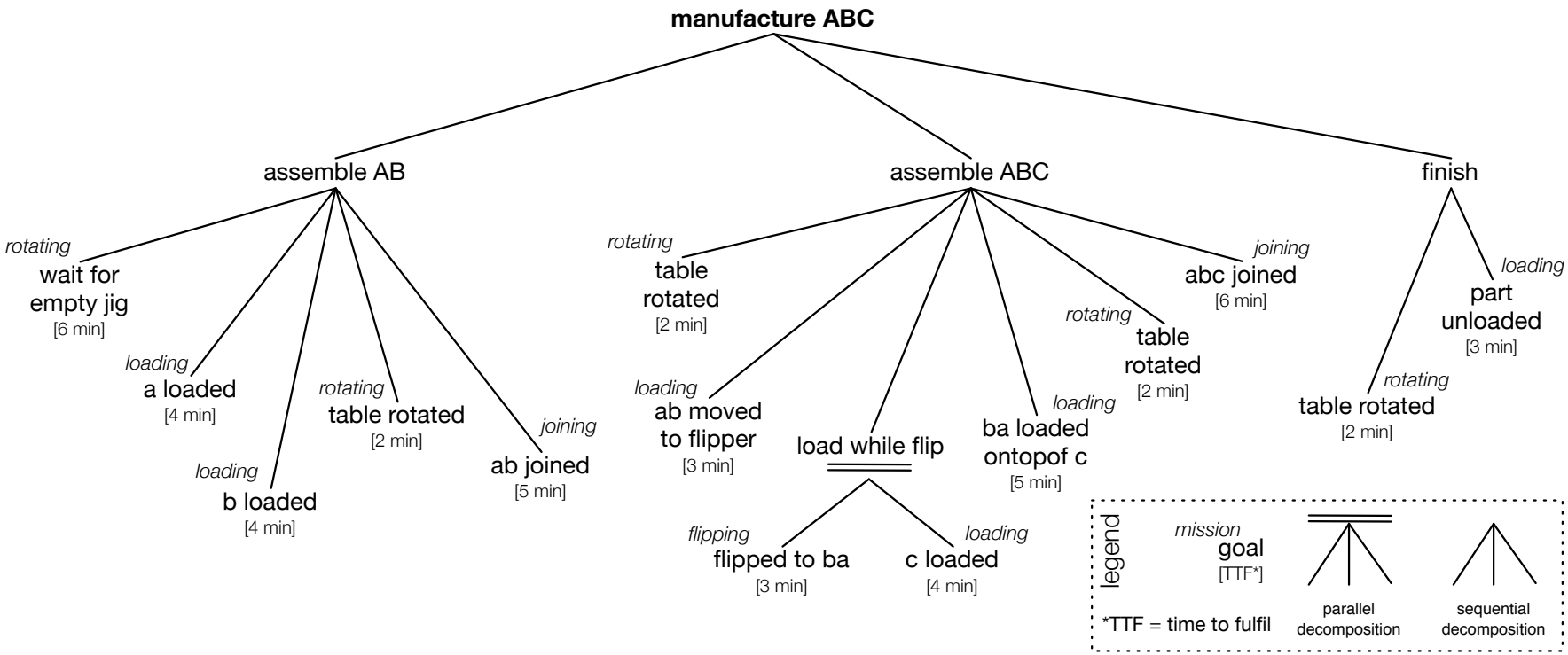
# The Social Plan II

- Much interagent communication was saved by delegating the coordination task to the organisation
- At run-time, the organisation will assign goals to agents at the appropriate times, taking into consideration the partial ordering of the goals to be achieved according to the functional specification
- Functional structure diagram also annotates each goal to be achieved with one of four different *missions* (*loading*, *joining*, *flipping*, and *rotating*)

# The Social Plan III

- The normative refers to these mission to determine which sets of goals the agents playing each of the four roles will be asked to achieve (by the organisation management infrastructure, at runtime)

# MOISE Organisation: Functional Specification



# MOISE Organisation: Normative Specification

| norm | role | mission |
|---|---|---|
| n1 | loader | *loading* |
| n2 | joiner | *joining* |
| n3 | rotator | *rotating* |
| n4 | flipper | *flipping* |

# 4.2 Agent Programs

# Plans to Facilitate Interaction with *M*OISE and CARTAGO

- JASON plans available with JACAMO which facilitate interaction with *M*OISE and CARTAGO when programming the agents
- Some such plans appear in file `common.asl`

## Example 4.1

Generic Plan for Agents that are part of an Organisation

```
// obligation to achieve a goal
+obligation(Ag,Norm,achieved(Scheme,Goal,Ag),Deadline) :
   .my_name(Ag) <-
      !Goal[scheme(Scheme)];
      lookupArtifact(Scheme,Id);
      goalAchieved(Goal)[artifact_id(Id)].
```

# Explanation of the Previous Plan

- That plan says that whenever the agent comes to believe that it has a new obligation towards an organisational goal `Goal` (note the use of Jason higher-order variables here), it just tries to achieve that goal
- If all goes well, the agent tells the organisation, through an ORA4MAS artifact, that the goal it was obliged to achive has been achieved (this is important so that the organisation can then delegate further goals to be achieved, possibly by other agents)

# Simple Controller Agents I

- In this application, the actual behaviour for agents "loader", "joiner", and "flipper" is to simply adopt its predetermined role and then do whatever it is asked to do
- For example, when the *M*OISE scheme determines the adoption of goal `!a_loaded`, the agent should just do the action `a_loaded` that activates the loading mechanism in the actual factory

# Simple Controller Agents II

- This is possible because the name of such operations in the artifact simulating the manufacturing cell is the same as the goal itself
- Artefact operations automatically become external actions for the Jason agent to use in plans
- This can be done in a generic way (through the use of the higher-order variable `G` below), for any organisational goal received

# Simple Controller Agents III

- Initially the agent joins the ORA4MAS workspace so as to take part in the organisation, then it also needs to focus on the ORA4MAS organisational artifact so as to automatically perceive information about the group such as newly created schemes
- The agent then adopts a role in the group (the group and specific role for each of the three agents using this code are specified as initial goals in the Jason project file)
- This is the complete code for the simpler agents:

# Simple Controller Agents IV

```
// Join the organisation and play a role in it
+!join_and_play(GroupName, RoleName)
  <- !in_ora4mas;
     lookupArtifact(GroupName, GroupId);
     focus(GroupId);
     adoptRole(RoleName)[artifact_id(GroupId)].

// Then, just do whatever told by the organisation
+!G[scheme(S)] <- G; .print("Doing ", G, " - Scheme ", S).
```

# The Rotator Agent I

- In the *M*OISE scheme for the manufacturing process, the rotator is assigned two different goals: to wait for an empty jig and to get the table rotated
- As a new order can be started at any time during the manufacturing of another, it is a rather difficult synchronisation problem that the rotator has to solve
- The next slides show the various plans needed to achieve these goals
- There are two simple prolog-like rules used to facilitate the plan contexts

# The Rotator Agent II

- They check the number of instances of the manufacturing scheme in *M*OISE so as to check if there are 1 or 2 concurrent orders being manufactured by this cell (each order is handled by one scheme instance)
- The name of the scheme that requested the achievement of a particular goal is annotated in the new goal events

# The Rotator Agent III

```
// rule to check if we have two concurrent orders (2 Moise schemes)
two_orders :- schemes(L) & .length(L)==2.
// or only one order so far
one_order :- schemes(L) & .length(L)==1.


// 1st organisational goal of the rotator (wait for empty jig)

// avoid conflicts when 2 orders are simultaneously waiting for empty jigs
+!wait_for_empty_jig[scheme(S1)] :
   .desire(wait_for_empty_jib[scheme(S2)]) & S1\==S2 <-
      .wait(500);                          // wait a bit
      !wait_for_empty_jig[scheme(S1)].     // and try again

// already got an empty jig
+!wait_for_empty_jig[scheme(S)] :
```

# The Rotator Agent IV

```
   jig_loader("empty") <-
      reserve_jig(S). // make sure another order doesn't get it too

// will have to wait until the jig at the loader end is empty
+!wait_for_empty_jig[scheme(S)] <-
      .wait({+jig_loader("empty")});  // wait until this event happens
      reserve_jig(S); // make sure empty jig is allocated to this order
      // if there are pending requests to rotate the table
      if (.desire(table_rotated[scheme(S)])) {
         // might need reconsidering which plan to use for rotating
         .drop_desire(table_rotated[scheme(S)]);
         !!table_rotated[scheme(S)];
      }.


// 2nd organisational goal of the rotator (rotate table)
```

# The Rotator Agent V

```
// Only 1 assembling task, rotate whenever asked
+!table_rotated : one_order <- table_rotated.

// Let it rotate if another job needs it and we're waiting for an empty jig
+!table_rotated :
   two_orders & .desire(wait_for_empty_jig) & not jig_loader("empty") <-
      table_rotated.

// If there are 2 concurrent assembling tasks, wait for both
// to want to rotate before actually rotating

// This is actually the second request to rotate
@tr[atomic] // both goals need to be considered achieved simultaneously
+!table_rotated[scheme(S1)] :
   two_orders & .desire(table_rotated[scheme(S2)]) & S1\==S2 <-
```

# The Rotator Agent VI

```
      table_rotated; // one rotation achieves both requests
      .succeed_goal(table_rotated[scheme(S2)]).

// The first attempt just waits, 2nd request releases both
+!table_rotated[scheme(S)] :
   two_orders <-
      .wait(1000); // wait a bit
      !table_rotated[scheme(S)]. // try again
```

# Cell Manager I

- The cell manager agent has mostly procedural code to create the simulation artifacts and initialise the organisation
- Other than that, it has only a few plans, the following one being the longest:

# Cell Manager II

## Example 4.2

```
// each order generates an instance of the Manufacture scheme
@op1[atomic] // needs to be an atomic operation: changing the no. of schemes
+order(N) :
   formationStatus(ok)[artifact_id(GrArtId)]
   & schemes(L) & .length(L)<=1  <- // no more than 1 order under way
      // wait until empty jig is correctly positioned at loader robor
      .concat("order", N, SchemeName);
      makeArtifact(SchemeName, "ora4mas.nopl.SchemeBoard",
         ["src/manufacture-os.xml", manufacture_schm, false, true], SchArtId);
      focus(SchArtId); // get all info about this Moise scheme
      addScheme(SchemeName)[artifact_id(GrArtId)].
```

# Cell Manager III

- The preceeding plan accepts at most two concurrent manufacturing orders, and creates the necessary ORA4MAS scheme artifact to handle a new (simulated) manufacturing order
- Focusssing on the scheme allowed the cell manager to automatically perceive the state of the scheme
- For example, it needs to know when the order has been completed so that a new one can be accepted
- This plan also needs to add the newly created scheme to the (well-formed) *M*OISE group

# 4.3 Environment Program

# Environment I

- A few environment artifacts were mentioned abova
- It also makes sense to use an artifact to manage instances of the *Contract-Net Protocol* (CNP), required in the scenario as a mechanism for load balancing (i.e. selecting the best assembly cell for a particular part assembling request)
- While in most cases artifacts required for particular tasks have to be developed using the CArtAgO API, in this case we do not need to do that as CArtAgO already offers an artifact-based implementation of CNP management

# Environment II

- Of course CNP can also be managed directly by agents, as in the example given in [Bordini et al., 2007b, Section 6.3]
- There are some advantages of using the artifact-based implementation in this case
- For example it reduces the amount of direct agent-to-agent communication required and allows the use of CNP in open multiagent systems: when agents join a CArtAgO workspace, they will be able to automatically perceive the available CNP instances and join in if they so wish

# Environment III

- Even though it is not necessary to program the artifact in this case, we show the code of one of the artifacts (the task board) just to illustrate the environment side of the system
- It also helps showing how artifacts are at a different level of abstracts as normal objects in Java
- The observable properties and operations automatically become percepts/action available to all agents that enter the shared workspace

# Environment IV

- In the code for the task board (available at `http://cartago.sourceforge.net`), agents use the `announce` operation on this artifact when they wish to start a new CNP instance for a particular task
- This artifact will then create another artifact to manage that particular instance of the CNP, with an observable property showing the task description (which again is accessible to any agents joining the workspace at runtime)

# Environment V

- It is in that newly created artifact that agents will be able to bid, and the agent being awarded the contract will be announced there too

```
public class TaskBoard extends Artifact {
  private int taskId;

  void init() {
    taskId = 0;
  }

  @OPERATION void announce(String taskDescr, int duration, OpFeedbackParam<String>
    taskId++;
    try {
      String artifactName = "cnp_board_"+taskId;
      makeArtifact(artifactName, "c4jexamples.ContractNetBoard",
```

# Environment VI

```
                    new ArtifactConfig(taskDescr,duration));
      defineObsProperty("task", taskDescr, artifactName);
      id.set(artifactName);
    } catch (Exception ex) {
      failed("announce_failed");
    }
  }

  @OPERATION void clear(String id) {
    String artifactName = "cnp_board_"+taskId;
    this.removeObsPropertyByTemplate("task", null, artifactName);
  }
}
```

# Some Final Comments

- The slides only included excerpts, although all the important parts of the code were covered
- Best approach to understand this example is to look at the complete (fully commented) code and run the system
- The working example for one manufacturing cell can be dowloaded from `http://www.inf.pucrs.br/r.bordini/WeissBookChapter13Ex`
- We leave as exercise to use the CNP artifacts for extending to multiple cells

# 5. Acknowledgements

# Acknowledgements

- Thanks to Jomi Hübner, Olivier Boissier, and Alessandro Ricci for some of the slides on JASON, *M*OISE, and CARTAGO.

# 6. References

## 5 References


Alechina, N., Dastani, M., Logan, B., and Meyer, J.-J. C. (2011).
Reasoning about Agent Deliberation.
*Autonomous Agents and Multi-Agent Systems*, 22(2):356–381.


Austin, J. L. (1962).
*How to Do Things with Words*.
Oxford University Press, London.


Bordini, R. H., Braubach, L., Dastani, M., Fallah-Seghrouchni, A. E., Gómez-Sanz, J. J., Leite, J., O'Hare, G. M. P., Pokahr, A., and Ricci, A. (2006).
A Survey of Programming Languages and Platforms for Multi-agent Systems.
*Informatica (Slovenia)*, 30(1):33–44.


Bordini, R. H., Dastani, M., Dix, J., and Fallah-Seghrouchni, A. E. (2011).
Preface.
*Autonomous Agents and Multi-Agent Systems*, 23(2):155–157.


Bordini, R. H., Dastani, M., Dix, J., and Seghrouchni, A. E. F. (2007a).
Preface – Special Issue on Programming Multiagent Systems.
*International Journal of Agent-Oriented Software Engineering*, 1(3/4).


Bordini, R. H., Hübner, J. F., and Wooldridge, M. (2007b).
*Programming Multi-agent Systems in AgentSpeak Using* JASON.
Wiley Series in Agent Technology. John Wiley & Sons.


Braubach, L. and Pokahr, A. (2011).

Addressing Challenges of Distributed Systems Using Active Components.
In Brazier, F., Nieuwenhuis, K., Pavlin, G., Warnier, M., and Badica, C., editors, *Intelligent Distributed Computing V - Proc 5th International Symposium on Intelligent Distributed Computing (IDC 2011)*, volume 382 of *Computational Intelligence*, pages 141–151. Springer.

Braubach, L., Pokahr, A., Moldt, D., and Lamersdorf, W. (2004).

Goal Representation for BDI Agent Systems.
In Bordini, R. H., Dastani, M., Dix, J., and Fallah-Seghrouchni, A. E., editors, *ProMAS*, volume 3346 of *Lecture Notes in Computer Science*, pages 44–65. Springer.

Clancey, W. J., Sierhuis, M., Kaskiris, C., and van Hoof, R. (2003).

Advantages of Brahms for Specifying and Implementing a Multiagent Human-Robotic Exploration System.
In Russell, I. and Haller, S. M., editors, *Proc. FLAIRS Conference*, pages 7–11. AAAI Press.

Collier, R., Dix, J., and Novák, P., editors (2011).

*Programming Multi-agent Systems, Revised Selected and Invited Papers of ProMAS 2010*, volume 6599 of *Lecture Notes in Computer Science*.
Springer.

Dastani, M. (2008).

2APL: A Practical Agent Programming Language.
*Autonomous Agents and Multi-Agent Systems*, 16(3):214–248.

Dastani, M., Fallah-Seghrouchni, A. E., Leite, J., and Torroni, P., editors (2010).

*Languages, Methodologies, and Development Tools for Multi-agent Systems, Second International Workshop, LADS 2009, Torino, Italy, September 7-9, 2009, Revised Selected Papers*, volume 6039 of *Lecture Notes in Computer Science*. Springer.

Dastani, M. and Gómez-Sanz, J. J. (2005).

Programming Multi-agent Systems.
*Knowledge Eng. Review*, 20(2):151–164.

Fisher, M. (1996).

Temporal Semantics for Concurrent MetateM.
*Journal of Symbolic Computation*, 22(5/6):627–648.

Fisher, M. (1997).

A Normal Form for Temporal Logics and its Applications in Theorem-Proving and Execution.
*Journal of Logic and Computation*, 7(4):429–456.

Fisher, M., Bordini, R. H., Hirsch, B., and Torroni, P. (2007).

Computational Logics and Agents: A Road Map of Current Technologies and Future Trends.
*Computational Intelligence*, 23(1):61–91.

Gâteau, B., Boissier, O., Khadraoui, D., and Dubois, E. (2005).

Moiseinst: An organizational model for specifying rights and duties of autonomous agents.
In *Third European Workshop on Multi-Agent Systems (EUMAS 2005)*, pages 484–485, Brussels Belgium.

Georgeff, M. P. and Lansky, A. L. (1987).

Reactive Reasoning and Planning.

In *AAAI*, pages 677–682.

Giacomo, G. D., Lespérance, Y., and Levesque, H. J. (2000).
ConGolog: A Concurrent Programming Language Based on the Situation Calculus.
*Journal of Artificial Intelligence*, 121(1-2):109–169.

Giacomo, G. D., Lespérance, Y., Levesque, H. J., and Sardinia, S. (2009).
IndiGolog: A High-Level Programming Language for Embedded Reasoning Agents.
In Bordini, R., Dastani, M., Dix, J., and Segrouchni, A. E. F., editors, *Multi-agent Programming: Languages, Tools and Applications*, pages 31–72. Springer.

Hannoun, M., Boissier, O., Sichman, J. S., and Sayettat, C. (2000).
MOISE: An organizational model for multi-agent systems.
In Monard, M. C. and Sichman, J. S., editors, *Proceedings of the International Joint Conference, 7th Ibero-American Conference on AI, 15th Brazilian Symposium on AI (IBERAMIA/SBIA'2000), Atibaia, SP, Brazil, November 2000*, LNAI 1952, pages 152–161, Berlin. Springer.

Hindriks, K. V. (2007).
Modules as Policy-Based Intentions: Modular Agent Programming in GOAL.
In Dastani, M., Fallah-Seghrouchni, A. E., Ricci, A., and Winikoff, M., editors, *ProMAS*, volume 4908 of *Lecture Notes in Computer Science*, pages 156–171. Springer.

Hindriks, K. V. and Roberti, T. (2009).
GOAL as a Planning Formalism.
In Braubach, L., van der Hoek, W., Petta, P., and Pokahr, A., editors, *MATES*, volume 5774 of *Lecture Notes in Computer Science*, pages 29–40. Springer.


Hübner, J. F., Boissier, O., and Bordini, R. H. (2010).
A normative organisation programming language for organisation management infrastructures.
In et al., J. P., editor, *Coordination, Organizations, Institutions and Norms in Agent Systems V*, volume 6069 of *LNAI*, pages 114–129. Springer.


Hübner, J. F., Boissier, O., Kitio, R., and Ricci, A. (2009).
Instrumenting Multi-Agent Organisations with Organisational Artifacts and Agents.
*Journal of Autonomous Agents and Multi-Agent Systems*.


Hübner, J. F., Sichman, J. S., and Boissier, O. (2002a).
A model for the structural, functional, and deontic specification of organizations in multiagent systems.
In Bittencourt, G. and Ramalho, G. L., editors, *Proceedings of the 16th Brazilian Symposium on Artificial Intelligence (SBIA'02)*, volume 2507 of *LNAI*, pages 118–128, Berlin. Springer.


Hübner, J. F., Sichman, J. S., and Boissier, O. (2002b).
MOISE+: Towards a Structural, Functional, and Deontic Model for MAS Organization.
In Castelfranchi, C. and Johnson, W. L., editors, *Proc. of International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS-02)*, pages 501–502. ACM Press.


Hübner, J. F., Sichman, J. S., and Boissier, O. (2006).
S-MOISE+: A middleware for developing organised multi-agent systems.
In Boissier, O., Dignum, V., Matson, E., and Sichman, J. S., editors, *Coordination, Organizations, Institutions, and Norms in Multi-Agent Systems*, volume 3913 of *LNCS*, pages 64–78. Springer.


Hübner, J. F., Sichman, J. S., and Boissier, O. (2007).

Developing Organised Multi-Agent Systems Using the MOISE+ Model: Programming Issues at the System and Agent Levels.
*Agent-Oriented Software Engineering*, 1(3/4):370–395.

Jordan, H. R., Treanor, J., Lillis, D., Dragone, M., Collier, R. W., and O'Hare, G. M. P. (2010).
AF-ABLE in the *Multi-Agent Programming Contest* 2009.
*Annals of Mathematics and Artificial Intelligence*, 59(3-4):389–409.

Lillis, D., Collier, R. W., Dragone, M., and O'Hare, G. M. P. (2009).
An Agent-Based Approach to Component Management.
In *Proc. 8th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 529–536.

Mascardi, V., Martelli, M., and Sterling, L. (2004).
Logic-Based Specification Languages for Intelligent Software Agents.
*Theory and Practice of Logic Programming*, 4(4):429–494.

Omicini, A., Ricci, A., and Viroli, M. (2008).
Artifacts in the A&A Meta-model for Multi-agent Systems.
*Autonomous Agents and Multi-Agent Systems*, 17(3).

Omicini, A., Ricci, A., Viroli, M., Castelfranchi, C., and Tummolini, L. (2004).
Coordination artifacts: Environment-based coordination for intelligent agents.
In *Proc. of the 3rd Int. Joint Conf. on Autonomous Agents and Multi-Agent Systems (AAMAS'04)*, volume 1, pages 286–293, New York, USA. ACM.


Omicini, A., Sardina, S., and Vasconcelos, W. W., editors (2011).
*Declarative Agent Languages and Technologies VIII - 8th International Workshop, DALT 2010, Toronto, Canada, May 10, 2010, Revised, Selected and Invited Papers*, volume 6619 of *Lecture Notes in Computer Science*. Springer.


Piunti, M., Ricci, A., Boissier, O., and Hubner, J. (2009).
Embodying organisations in multi-agent work environments.
In *IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT 2009)*, Milan, Italy.


Piunti, M., Ricci, A., Braubach, L., and Pokahr, A. (2008).
Goal-directed interactions in artifact-based mas: Jadex agents playing in CARTAGO environments.
In *Proc. of the 2008 IEEE/WIC/ACM Int. Conf. on Web Intelligence and Intelligent Agent Technology (IAT'08)*, volume 2. IEEE Computer Society.


Pokahr, A., Braubach, L., and Lamersdorf, W. (2005).
Jadex: A BDI Reasoning Engine.
In Bordini, R. H., Dastani, M., Dix, J., and Fallah-Seghrouchni, A. E., editors, *Multi-Agent Programming*, volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*, pages 149–174. Springer.


Rao, A. S. (1996).
AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language.
In de Velde, W. V. and Perram, J. W., editors, *Proc. 7th European Workshop on Modelling Autonomous Agents in a Multi-agent World (MAAMAW)*, volume 1038 of *Lecture Notes in Computer Science*, pages 42–55. Springer.


Rao, A. S. and Georgeff, M. P. (1995).
BDI Agents: From Theory to Practice.

In Lesser, V. R. and Gasser, L., editors, *Proc. First International Conference on Multiagent Systems (ICMAS)*, pages 312–319. The MIT Press.

Ricci, A., Piunti, M., Acay, L. D., Bordini, R., Hübner, J., and Dastani, M. (2008).
Integrating artifact-based environments with heterogeneous agent-programming platforms.
In *Proceedings of 7th International Conference on Agents and Multi Agents Systems (AAMAS08)*.

Ricci, A., Piunti, M., Viroli, M., and Omicini, A. (2009a).
Environment programming in CArtAgO.
In *Multi-Agent Programming: Languages,Platforms and Applications,Vol.2*. Springer.

Ricci, A., Piunti, M., Viroli, M., and Omicini, A. (2009b).
Environment Programming in CArtAgO.
In Bordini, R. H., Dastani, M., Dix, J., and El Fallah-Seghrouchni, A., editors, *Multi-agent Programming: Languages, Platforms and Applications, Vol. 2*, pages 259–288. Springer.

Ricci, A., Santi, A., and Piunti, M. (2010).
Action and perception in multi-agent programming languages: From exogenous to endogenous environments.
In *In Proceedings of International Workshop on Programming Multi-Agent Systems (ProMAS-8)*.

Ricci, A., Viroli, M., and Omicini, A. (2007a).
The A&A programming model & technology for developing agent environments in MAS.
In Dastani, M., El Fallah Seghrouchni, A., Ricci, A., and Winikoff, M., editors, *Programming Multi-Agent Systems*, volume 4908 of *LNAI*, pages 91–109. Springer Berlin / Heidelberg.

Ricci, A., Viroli, M., and Omicini, A. (2007b).

CArtAgO: A framework for prototyping artifact-based environments in MAS.
In Weyns, D., Parunak, H. V. D., and Michel, F., editors, *Environments for MultiAgent Systems III*, volume 4389 of *LNAI*, pages 67–86. Springer.
3rd International Workshop (E4MAS 2006), Hakodate, Japan, 8 May 2006. Selected Revised and Invited Papers.

Searle, J. R. (1969).

*Speech Acts: An Essay in the Philosophy of Language*.
Cambridge University Press, Cambridge.

Stocker, R., Sierhuis, M., Dennis, L. A., Dixon, C., and Fisher, M. (2011).

A Formal Semantics for Brahms.
In *Proc. 12th International Workshop on Computational Logic in Multi-agent Systems (CLIMA)*, volume 6814 of *Lecture Notes in Computer Science*, pages 259–274. Springer.

van Putten, B.-J., Dignum, V., Sierhuis, M., and Wolfe, S. R. (2008).

OperA and Brahms: A Symphony?
In *Proc. 9th International Workshop on Agent-Oriented Software Engineering (AOSE)*, volume 5386 of *Lecture Notes in Computer Science*, pages 257–271. Springer.

van Riemsdijk, B., Dastani, M., and Meyer, J.-J. C. (2005).

Semantics of Declarative Goals in Agent Programming.
In *Proc. 4th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 133–140. ACM.