PROGRAMMING AGENTS AND MULTI-AGENT SYSTEMS

Agent-Oriented Computing Course

Alessandro Ricci
a.ricci@unibo.it
DEIS - Alma Mater Studiorum Università di Bologna
FROM AOSE
TO AGENT AND MULTI-AGENT SYSTEM
PROGRAMMING
• From design to implementation and deployment
  - how to develop a software agent? A multi-agent system? or
  - how to develop a software system in terms of agents, as a multi-agent system?
AGENT-ORIENTED PROGRAMMING DEVELOPMENT TOOLS
AGENT-ORIENTED PROGRAMMING DEVELOPMENT TOOLS

- Good surveys can be found in [MAP1, MAP2]
  - multi-agent programs development
- **agent programming languages**
  - the most part developed in the context of AI / Distributed AI
  - examples include AgentSpeak(L)/Jason, 3APL, 2APL, GOAL, AgentFactory, ConGolog, ...
- **agent programming frameworks**
  - developed and used in particular in the context of AOSE
  - providing API based on mainstream programming languages and technologies
  - examples include JADE, JADEX, simpA...
- multi-agent program deployment and execution
  - platforms, middleware, infrastructures
MAIN DIMENSIONS
MAIN DIMENSIONS

• Agent-Oriented Programming (PART 1)
  - how to program individual agents
MAIN DIMENSIONS

- **Agent-Oriented Programming (PART 1)**
  - how to program individual agents

- **Multi-Agent Programming (PART 2)**
  - how to program the *ensemble* of agents
    - interaction, communication, coordination, organisation
  - focus in particular on
    - Coordination Models, Languages and Technologies (PART 2.1)
    - Environment Programming (PART 2.2)
PART 1: AGENT-ORIENTED PROGRAMMING
AGENT-ORIENTED PROGRAMMING
AGENT-ORIENTED PROGRAMMING

• Programming individual agents
AGENT-ORIENTED PROGRAMMING

• Programming individual agents
• agent = architecture + program
  - agent architecture = software framework within which an agent program runs
    • including sensors and actuators to interact with the environment, specific structures to manage the agent state, tasks and so on
  - an example: BDI architecture
    • agent program = describe how to perform the task(s) for which the agent has been designed
AGENT-ORIENTED PROGRAMMING

- Programming individual agents
- \texttt{agent} = \texttt{architecture} + \texttt{program}
  - \texttt{agent architecture} = software framework within which an agent program runs
    - including sensors and actuators to interact with the environment, specific structures to manage the agent state, tasks and so on
  - an example: BDI architecture
    - \texttt{agent program} = describe how to perform the task(s) for which the agent has been designed
- So we write the program that will direct the agent behaviour...
  - ...but depending on the architecture, much of what the agent effectively does is determined by the architecture itself
- without the programmer having to worry about it
AGENT-ORIENTED PROGRAMMING LANGUAGES
AGENT-ORIENTED PROGRAMMING LANGUAGES

• First AOP language introduced by Shoham in 1993, called Agent-0 [Sho93]
  - basic aim: introducing a post-OO programming paradigm based on a cognitive and societal view of computation
AGENT-ORIENTED PROGRAMMING LANGUAGES

• First AOP language introduced by Shoham in 1993, called Agent-0 [Sho93]
  - basic aim: introducing a post-OO programming paradigm based on a cognitive and societal view of computation
• Many languages have been developed since then
  - almost all in the context of AI and Distributed AI
  - possible classification based on the underlying architecture
    • based on practical reasoning and BDI
      • ex: AgentSpeak(L)/Jason, 3APL, 2APL, GOAL, AgentFactory, ...
    • logic-based, based on a purely deliberative architecture
      • ex: Concurrent MetateM, ConGolog, Minerva, ...
AGENT-ORIENTED PROGRAMMING LANGUAGES

- First AOP language introduced by Shoham in 1993, called Agent-0 [Sho93]
  - basic aim: introducing a post-OO programming paradigm based on a cognitive and societal view of computation
- Many languages have been developed since then
  - almost all in the context of AI and Distributed AI
  - possible classification based on the underlying architecture
    - based on practical reasoning and BDI
      - ex: AgentSpeak(L)/Jason, 3APL, 2APL, GOAL, AgentFactory,...
    - logic-based, based on a purely deliberative architecture
      - ex: Concurrent MetateM, ConGolog, Minerva,...
- In this course we focus on AgentSpeak(L)/Jason as reference case
AgentSpeak(L) AND Jason
AgentSpeak(L) AND Jason

- **AgentSpeak(L)**
  - abstract language used for describing and programming BDI agents [Rao96]
  - inspired to PRS (Procedural Reasoning System) architecture and BDI logics
AgentSpeak(L) AND Jason

- **AgentSpeak(L)**
  - abstract language used for describing and programming BDI agents [Rao96]
  - inspired to PRS (Procedural Reasoning System) architecture and BDI logics

- **Jason** [BH06][BHW07]
  - AgentSpeak(L) extension to make it a *practical* agent programming language
  - concrete language and customisable Java-based interpreter / platform
    - can be downloaded at [http://jason.sourceforge.net](http://jason.sourceforge.net)
    - open-source
RECALLING BDI-ABSTRACT ARCHITECTURE

Agent

Sensors

BRF

Beliefs

Generate Options

Desires

Intentions

Action

Effectors

Filter

A. Ricci

Programming Agents and MAS - Scuola Dottorato in Ing. e Scienza dell'Informazione
REASONING CYCLE OF A BDI AGENT

1. \( B \leftarrow B_0; \) /* \( B_0 \) are initial beliefs */
2. \( I \leftarrow I_0; \) /* \( I_0 \) are initial intentions */
3. while true do
4.     get next percept \( \rho \) via sensors;
5.     \( B \leftarrow \text{brf}(B, \rho); \)
6.     \( D \leftarrow \text{options}(B, I); \)
7.     \( I \leftarrow \text{filter}(B, D, I); \)
8.     \( \pi \leftarrow \text{plan}(B, I, Ac); \) /* \( Ac \) is the set of actions */
9.     while not (empty(\( \pi \)) or succeeded(\( I, B \)) or impossible(\( I, B \))) do
10.        \( \alpha \leftarrow \text{first element of } \pi; \)
11.        \( \text{execute}(\alpha); \)
12.        \( \pi \leftarrow \text{tail of } \pi; \)
13.        observe environment to get next percept \( \rho; \)
14.        \( B \leftarrow \text{brf}(B, \rho); \)
15.        if \( \text{reconsider}(I, B) \) then
16.            \( D \leftarrow \text{options}(B, I); \)
17.            \( I \leftarrow \text{filter}(B, D, I); \)
18.        end-if
19.        if not \( \text{sound}(\pi, I, B) \) then
20.            \( \pi \leftarrow \text{plan}(B, I, Ac) \)
21.        end-if
22.     end-while
23. end-while
AgentSpeak(L) AGENTS: MAIN CONCEPTS
AgentSpeak(L) AGENTS: MAIN CONCEPTS

• Beliefs
  - information the agent has about the world
  - including the state of the environment, other agents, itself
AgentSpeak(L) AGENTS: MAIN CONCEPTS

• Beliefs
  - information the agent has about the world
  - including the state of the environment, other agents, itself

• Goals
  - something that the agent wants to achieve
AgentSpeak(L) AGENTS: MAIN CONCEPTS

• **Beliefs**
  - information the agent has about the world
  - including the state of the environment, other agents, itself

• **Goals**
  - something that the agent wants to achieve

• **Plans**
  - course of action or "recipe" to achieve some goal
AgentSpeak(L) AGENTS
ARCHITECTURE

A. Ricci

Programming Agents and MAS - Scuola Dottorato in Ing. e Scienza dell'Informazione
MAIN ARCHITECTURAL COMPONENTS
MAIN ARCHITECTURAL COMPONENTS

• Belief Base
  - set of current beliefs
MAIN ARCHITECTURAL COMPONENTS

- **Belief Base**
  - set of current beliefs

- **Plan Library**
  - set of available plans
  - this is the agent program, actually
MAIN ARCHITECTURAL COMPONENTS

- **Belief Base**
  - set of current beliefs

- **Plan Library**
  - set of available plans
  - this is the agent program, actually

- **Set of Events**
  - agent/environment changes that trigger and drive agent behaviour
MAIN ARCHITECTURAL COMPONENTS

- **Belief Base**
  - set of current beliefs

- **Plan Library**
  - set of available plans
  - this is the agent program, actually

- **Set of Events**
  - agent/environment changes that trigger and drive agent behaviour

- **Intentions**
  - set of instantiated/running plans
Jason AGENT ARCHITECTURE

1. Percepts
   - perceive

2. BUF
   - BRF
   - External Events
   - Beliefs

3. Messages
   - checkMail
   - Messages
   - SocAcc
   - Suspended Intentions
     - (Actions and Msgs)

4. SocAcc
   - Beliefs to Add and Delete

5. Events
   - Selected Event
   - Relevant Plans
   - Applicable Plans
   - Intended Means
   - Selected Intention

6. Unify Event
   - Plans
   - Beliefs
   - New Plan
   - Push Plan
   - New Intention

7. Check Context
   - Actions
   - New Intention
   - Suspended Intentions
   - (Actions and Msgs)

8. Intents
   - Actions
   - New Intention
   - New Intentions
   - Suspended Intentions
   - (Actions and Msgs)

9. Execute Intention
   - New Intention
   - Intents
   - Actions
   - New Intention
   - New Actions

10. Plan Library
    - newMsg
    - sendMsg
    - new

A. Ricci
Programming Agents and MAS - Scuola Dottorato in Ing. e Scienza dell'Informazione
JASON ARCHITECTURE - STAGES
JASON ARCHITECTURE - STAGES

- Steps
  1. Perceiving the Environment
  2. Updating the Belief Base
  3. Receiving Communication from Other Agents
  4. Selecting `Socially Acceptable' Messages
  5. Selecting an Event
  6. Retrieving all Relevant Plans
  7. Determining the Applicable Plans
  8. Selecting one Applicable Plan
  9. Selecting an Intention for Further Execution
  10. Executing one step of an Intention
JASON ARCHITECTURE - STAGES

- Steps
  - 1 Perceiving the Environment
  - 2 Updating the Belief Base
  - 3 Receiving Communication from Other Agents
  - 4 Selecting `Socially Acceptable' Messages
  - 5 Selecting an Event
  - 6 Retrieving all Relevant Plans
  - 7 Determining the Applicable Plans
  - 8 Selecting one Applicable Plan
  - 9 Selecting an Intention for Further Execution
  - 10 Executing one step of an Intention
A JASON AGENT PROGRAM AT A GLANCE

/* Initial Beliefs */
likes(radiohead).
phone_number(covo,"05112345")

/* Plans */

@monitor_concerts
+concert(Artist, Date, Venue)
  : likes(Artist)
  <- !book_tickets(Artist, Date, Venue).

@book_tickets
+!book_tickets(A,D,V)
  : not busy(phone)
  <- ?phone_number(V,N); /* Test Goal to Retrieve a Belief */
    !call(N);
    ...;
    !choose_seats(A,D,V).
A JASON AGENT PROGRAM AT A GLANCE

- Initial set of beliefs and goals + set of plans

```jason
/* Initial Beliefs */
likes(radiohead).
phone_number(covo,"05112345")

/* Plans */

@monitor_concerts
+concert(Artist, Date, Venue)
  : likes(Artist)
  <- !book_tickets(Artist, Date, Venue).

@book_tickets
+!book_tickets(A,D,V)
  : not busy(phone)
  <- ?phone_number(V,N); /* Test Goal to Retrieve a Belief */
    !call(N);
    ...;
    !choose_seats(A,D,V).
```
BELIEFS
BELIEFS

- Represented by literals
  - logic predicates or their negation, expressing a property of an object/individual or a relationship among objects/individuals
  - examples:
    - `temperature(reactor, 80)`
      - the agent currently believes that the temperature of the reactor is 80
    - `~raining`
      - the agent currently believes that it is not raining
BELIEFS

- Represented by literals
  - logic predicates or their negation, expressing a property of an object/individual or a relationship among objects/individuals
  - examples:
    - temperature(reactor, 80)
    - the agent currently believes that the temperature of the reactor is 80
    - ~raining
    - the agent currently believes that it is not raining
- in Jason, beliefs are used also to keep track of mental notes taken by the agent itself, not related to the environment
- operators +bel, -bel, ++bel can be used in plans to add, remove, replace a mental node, where bel is a ground literal
- e.g. +index(0)
GOALS
GOALS

- Distinguishing concept and feature of cognitive/intelligent agent programming languages
GOALS

• Distinguishing concept and feature of cognitive/intelligent agent programming languages
• Generally speaking, they represent properties of the states of the world that the agent wishes to bring about
  - by having a goal $g$, the agent is committed to act so as to change the world to a state in which the agent will, by sensing the environment, believes that $g$ is true
GOALS

• Distinguishing concept and feature of cognitive/intelligent agent programming languages

• Generally speaking, they represent properties of the states of the world that the agent wishes to bring about
  - by having a goal $g$, the agent is committed to act so as to change the world to a state in which the agent will, by sensing the environment, believes that $g$ is true

• Two goal types in AgentSpeak(L)/Jason
  - achievement goals
  - test goals
ACHIEVEMENT GOALS
ACHIEVEMENT GOALS

- Expressed by $!g$, where $g$ is a literal
  - the agent has the goal of achieving a certain state of affairs in which the agent will believe $g$
- example: $!own(house)$
ACHIEVEMENT GOALS

• Expressed by \( !g \), where \( g \) is a literal
  - the agent has the goal of achieving a certain state of affairs in which the agent will believe \( g \)
    • example: \( !\text{own(\text{house})} \)

• Two subtypes
  - **procedural** goals
    • the goal name is similar to the name of a procedure in a traditional programming language
    • e.g. \( !\text{clean(\text{house})} \)
  - **declarative** goals
    • symbolic representation of a state of affair that the agent has to achieve
    • e.g. \( !\text{cleaned(\text{house})} \)
TEST GOALS
TEST GOALS

- Expressed by $?g$, where $g$ is a literal
  - the agent has the goal to retrieve some information that is available in the agent's belief base
  - example: $?bank\_balance(X)$
PLANS

- Plans represent agent’s means to achieve goals (their know-how)
  - the set of plans constitute the agent programs
- An AgentSpeak(L) plan has the following general structure:

  ```
  @planlabel
  triggering_event : context <- body.
  ```

- where:
  - the (optional) plan label can be specified to identify the plan
  - the **triggering event** denotes the kind of events that the plan is meant to handle
  - the **context** represents the circumstances in which the plan can be used
    - logic expression over the belief base
  - the **body** is a sequence of actions to be performed to handle the event
    - sequence of formulae to be executed
EXAMPLES OF PLANS:
BACKGROUND CONTEXT
EXAMPLES OF PLANS: BACKGROUND CONTEXT

• Abstract version of a Mars exploration scenario: a typical day of activity of an autonomous Mars rover [BH07]
EXAMPLES OF PLANS: BACKGROUND CONTEXT

- Abstract version of a Mars exploration scenario: a typical day of activity of an autonomous Mars rover [BH07]
- Typical instructions sent to the rover by the ground team:
  1. Back up to the rock named Souffle
  2. Place the arm with the spectrometer on the rock
  3. Do extensive measurements on the rock surface
  4. Perform a long traverse to another rock
EXAMPLES OF PLANS: BACKGROUND CONTEXT

- Abstract version of a Mars exploration scenario: a typical day of activity of an autonomous Mars rover [BH07]
- Typical instructions sent to the rover by the ground team:
  1. Back up to the rock named Souffle
  2. Place the arm with the spectrometer on the rock
  3. Do extensive measurements on the rock surface
  4. Perform a long traverse to another rock
- It turned out that the robot was not correctly positioned, so scientific data was lost
EXAMPLES OF PLANS:
BACKGROUND CONTEXT

• Abstract version of a Mars exploration scenario: a typical day of activity of an autonomous Mars rover [BH07]

• Typical instructions sent to the rover by the ground team:
  1. Back up to the rock named Souffle
  2. Place the arm with the spectrometer on the rock
  3. Do extensive measurements on the rock surface
  4. Perform a long traverse to another rock

• It turned out that the robot was not correctly positioned, so scientific data was lost

• Green patches on rocks indicate good science opportunity
EXAMPLES OF PLANS: BACKGROUND CONTEXT

- Abstract version of a Mars exploration scenario: a typical day of activity of an autonomous Mars rover [BH07]
- Typical instructions sent to the rover by the ground team:
  1. Back up to the rock named Souffle
  2. Place the arm with the spectrometer on the rock
  3. Do extensive measurements on the rock surface
  4. Perform a long traverse to another rock
- It turned out that the robot was not correctly positioned, so scientific data was lost
- Green patches on rocks indicate good science opportunity
- Batteries only work while there is sunlight (“sol” is a Martian day)
EXAMPLES OF PLANS:
BACKGROUND CONTEXT

- Abstract version of a Mars exploration scenario: a typical day of activity of an autonomous Mars rover [BH07]
- Typical instructions sent to the rover by the ground team:
  1. Back up to the rock named Souffle
  2. Place the arm with the spectrometer on the rock
  3. Do extensive measurements on the rock surface
  4. Perform a long traverse to another rock
- It turned out that the robot was not correctly positioned, so scientific data was lost
- Green patches on rocks indicate good science opportunity
- Batteries only work while there is sunlight (“sol” is a Martian day)
- Detailed program used in the experiments had 25 plans
EXAMPLES OF PLANS (1/2)

@plan_one
+green_patch(Rock) : not battery_charge(low)
  <- ?location(Rock,Coordinates);
    !traverse(Coordinates);
    !examine(Rock).

@traverse_plan1
+!traverse(Coords) : safe_path(Coords)
  <- move_towards(Coords).

@traverse_plan1
+!traverse(Coords) : not safe_path(Coords)
  <- ...
EXAMPLES OF PLANS (2/2)

@examine_plan1
+!examine(Rock) : correctly_positioned(Rock)
  <- place_spectrometer(Rock);
  !extensive_measurements(Rock).

@examine_plan2
+!examine(Rock) :
  not correctly_positioned(Rock)
  <- !correctly_positioned(Rock);
  !examine(Rock).
PLAN SELECTION AND INSTANTIATION SEMANTICS
PLAN SELECTION AND INSTANTIATION SEMANTICS

- Given an event, all the *applicable* plans that are *relevant* for the event are instantiated, creating *new intentions*
  - *relevant*
    - the triggering event matches the event
  - *applicable*
    - the context is evaluated to true
TRIGGERING EVENTS
TRIGGERING EVENTS

• Triggering events express changes in the agent's mental attitudes that can be important for the agent program

- changes in the belief-base
  • Belief addition \( +b \) / Belief deletion \( -b \)
  • Belief change is modelled as deletion followed by addition
  • e.g.: \( +\)green_patch(Rock)
  • = the agent perceived a green patch on the rock

- changes in the goal-base
  • achievement-goal addition \( +!g \) / achievement-goal deletion \( -!g \)
  • test-goal addition \( +?g \) / test-goal deletion \( -?g \)
  • e.g.: \( +!\)examine(Rock)
  • = a new goal \( +!\)examine(Rock) has been added to the goal-base
REACTIVE AND PRO-ACTIVE BEHAVIOUR

• By specifying plans triggered by changes in beliefs we realise a *reactive behaviour*

\[
\text{@plan_one}
\begin{align*}
+ \text{green_patch(Rock)} &: \text{not battery_charge(low)} \\
\text{<- ?location(Rock,Coordinates);} \\
\text{!traverse(Coordinates);} \\
\text{!examine(Rock).}
\end{align*}
\]

• By specifying plans triggered by changes in the goal-base we realise a *pro-active behaviour*

\[
\text{@examine_plan1}
\begin{align*}
+ \text{!examine(Rock)} &: \text{correctly_positioned(Rock)} \\
\text{<- place_spectrometer(Rock);} \\
\text{!extensive measurements(Rock).}
\end{align*}
\]

\[
\text{@examine_plan2}
\begin{align*}
+ \text{!examine(Rock)} &: \\
\text{not correctly_positioned(Rock)} \\
\text{<- !correctly_positioned(Rock);} \\
\text{!examine(Rock).}
\end{align*}
\]
STRUCTURING PLANS

- Complex plans can be structured by including goals \(!g, ?g\) in the plan body
  - the plan is *suspended* until the specified goal(s) are achieved

- example:

  ```
  +event : some_context(X)
  <- action1;
  !my_subgoal1;
  action2;
  ?my_subgoal2;
  action3.
  ```

- action2 is executed only after that the achievement goal \(!my_subgoal1\) has been achieved, action3 is executed only after the test goal \(?my_subgoal2\) has been achieved

- Possibility to avoid sequential behaviour by using the operator \(!!g\) to achieve some goal creating a new intention:

  ```
  +event : some_context(X)
  <- action1;
  !!my_parallel_goal1;
  action2.
  ```

  - the execution of action2 is independent from the new intention executed to achieve \(!my_parallel_goal\)
ACTIONS
ACTIONS

- What the agent is capable of doing
ACTIONS

• What the agent is capable of doing
• Represented by predicates
  - ground parameters + (possibly) variables to be bound by action execution, representing *action feedbacks*
  - success/failure semantics
    • action failure causes plan failure
**ACTIONS**

- What the agent is capable of doing
- Represented by predicates
  - ground parameters + (possibly) variables to be bound by action execution, representing *action feedbacks*
  - success/failure semantics
    - action failure causes plan failure
- Two basic types
  - **external** = affecting the environment
    - ex: `switch_on(light45), move_towards(1,2)`...
  - **internal** = affecting the agent state (Jason extension)
    - manipulate/inspect the goal base, intention stacks
      - `.drop_intention(go(1,2)), .desire(go(1,2))`...
    - general-purpose computing functionalities
      - `.length([1,2,3], Len), .findall(X, a(X), L)`...
    - distinguished syntactically from ext. actions by having a dot “.” in the action name
COMMUNICATIVE ACTIONS
COMMUNICATIVE ACTIONS

- Special kind of actions
  - acts to communicate with other agents
  - based on *speech act* theory (Austin [Au62], Searle [Sea69])
COMMUNICATIVE ACTIONS

• Special kind of actions
  - acts to communicate with other agents
  - based on **speech act** theory (Austin [Au62], Searle [Sea69])

• **Agent Communication Language**
  - specific languages defining the set of speech act types
    • examples of speech acts: tell, ask, achieve, ...
  - giving an explicit high-level semantics
    • for instance based on the BDI model
      • if A tells B that $\Phi$, then B believes $\Phi$
COMMUNICATIVE ACTIONS

• Special kind of actions
  - acts to communicate with other agents
  - based on speech act theory (Austin [Au62], Searle [Sea69])

• Agent Communication Language
  - specific languages defining the set of speech act types
  • examples of speech acts: tell, ask, achieve, ...
  • giving an explicit high-level semantics
    • for instance based on the BDI model
      • if A tells B that Phi, then B believes Phi

• Jason action: .send(r, ilf, pc)
  - where ilf in
    • {tell, untell, achieve, unachieve, tellHow, untellHow, askIf, askOne, askAll, askHow}
THE ENVIRONMENT SIDE
THE ENVIRONMENT SIDE

- In actual deployment, there will normally be a real-world external environment where the MAS will be situated
  - target of the actions and source of the percepts
  - the agent architecture needs to be customised to get perceptions and act on such environment
THE ENVIRONMENT SIDE

- In actual deployment, there will normally be a real-world external environment where the MAS will be situated
  - target of the actions and source of the percepts
  - the agent architecture needs to be customised to get perceptions and act on such environment
- However, it is often useful to have a simulated environment (e.g., to test the MAS)
  - *Jason* provides a basic Java-based API to implements simulated environments
    - Environment base class
      - methods to specify action behaviour, including percepts generation
THE ENVIRONMENT SIDE

• In actual deployment, there will normally be a real-world external environment where the MAS will be situated
  • target of the actions and source of the percepts
  • the agent architecture needs to be customised to get perceptions and act on such environment
• However, it is often useful to have a simulated environment (e.g., to test the MAS)
  - Jason provides a basic Java-based API to implements simulated environments
  • Environment base class
    • methods to specify action behaviour, including percepts generation
• Actually in “Environment Programming” part we will go beyond this view
  - exploiting the environment as a first-class abstraction for MAS designers and developers, not only for simulations
ENVIRONMENT API

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)
...

UserEnvironment
+ init(String[] args)
+ executeAction(String agName, Structure action): boolean
ENVIRONMENT API EXAMPLE

```java
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;
    }

    // ...
DEFINING THE MAS CONFIGURATION

- Jason has a simple language for defining multi-agent program/system configuration
  - specifying the initial set of agents
  - each agent runs its own AgentSpeak(L)/Jason interpreter,
  - an environment, implemented by a Java class

```java
MAS MyProgram {

    environment: MyEnv // environment defined by the MyEnv.class

    agents:

        eugenio Explorer; // one agent called eugenio, agent program: Explorer.asl
        my_ant Ant #100; // 100 instances of Ant agents, called ant0, ant1, ...

...}
```
A COMPLETE AGENT PROGRAM
- DOMESTIC ROBOT SCENARIO -
A COMPLETE AGENT PROGRAM
- DOMESTIC ROBOT SCENARIO -

• Example from [RHW07]

  “A domestic robot has the goal of serving beer to its owner. Its mission is quite simple, it just receives some beer requests from the owner, goes to the fridge, takes out a bottle of beer, and brings it back to the owner. However, the robot should also be concerned with the beer stock (and eventually order more beer using the supermarket’s home delivery service) and some rules hard-wired into the robot by the Department of Health (in this example this rule defines the limit of daily beer consumption).”
A COMPLETE AGENT PROGRAM
- DOMESTIC ROBOT SCENARIO -

• Example from [RHW07]
  - “A domestic robot has the goal of serving beer to its owner. Its mission is quite simple, it just receives some beer requests from the owner, goes to the fridge, takes out a bottle of beer, and brings it back to the owner. However, the robot should also be concerned with the beer stock (and eventually order more beer using the supermarket’s home delivery service) and some rules hard-wired into the robot by the Department of Health (in this example this rule defines the limit of daily beer consumption).”

• Implementation available in Jason distribution
DESIGN SKETCH IN PROMETHEUS

- robot
  - open(fridge)
  - get(beer)
  - close(fridge)
  - hand_in(beer)
  - move_towards(Place)
  - achieve: order(beer, N)
  - tell: delivered(beer, N, OrderId)

- owner
  - achieve: has(owner, beer)
  - tell: toomuch(beer)

- stock(beer, N)
  - at(robot, Place)

- sip(beer)

- supermarket
  - deliver(beer, N)
OWNER AGENT

!get(beer). // initial goal

/* Plans */

@g
+!get(beer) : true
  <- .send(robot, achieve, has(owner,beer)).

@h1
+has(owner,beer) : true
  <- !drink(beer).

@h2
-has(owner,beer) : true
  <- !get(beer).

// while I have beer, sip
@d1
+!drink(beer) : has(owner,beer)
  <- sip(beer);
  !drink(beer).

@d2
+!drink(beer) : not has(owner,beer)
  <- true.

+msg(M)[source(Ag)] : true
  <- .print("Message from ",Ag," : ",M);
  -msg(M).
SUPER-MARKET AGENT

last_order_id(1). // initial belief

// plan to achieve the the goal "order" from agent Ag
+!order(Product,Qtd)[source(Ag)] : true
  <- ?last_order_id(N);
    OrderId = N + 1;
    +last_order_id(OrderId);
    deliver(Product, Qtd);
  .send(Ag, tell, delivered(Product, Qtd, OrderId)).
/* Initial beliefs */

// initially, I believe that there are some beers in the fridge
available(beer, fridge).

// my owner should not consume more than 10 beers a day :-)
limit(beer, 10).

/* Rules */
too_much(B) :-
    .date(YY, MM, DD) &
    .count(consumed(YY, MM, DD, _, _, _, B), QtdB) &
    limit(B, Limit) &
    QtdB > Limit.
/* Plans */
@h1
+!has(owner,beer)  
  : available(beer,fridge) & not too_much(beer)  
  <- !at(robot,fridge);  
      open(fridge);  
      get(beer);  
      close(fridge);  
      !at(robot,owner);  
      hand_in(beer);  
  // remember that another beer will be consumed  
  .date(YY,MM,DD); .time(HH,NN,SS);  
  +consumed(YY,MM,DD,HH,NN,SS,beer).

@h2
+!has(owner,beer)  
  : not available(beer,fridge)  
  <- .send(supermarket, achieve, order(beer,5));  
      !at(robot,fridge). // go to fridge and wait there.

@h3
+!has(owner,beer)  
  : too_much(beer) & limit(beer,L)  
  <- .concat("The Department of Health does not allow me ",  
             "to give you more than ", L,  
             " beers a day! I am very sorry about that!",M);  
      .send(owner,tell,msg(M)).
ROBOT AGENT: PLANS 2/2

@m2
+!at(robot,P) : not at(robot,P)
   <- move_towards(P);
     !at(robot,P).

// when the supermarket finishes the order, try the 'has'
// goal again
@a1
+delivered(beer,Qtd,OrderId)[source(supermarket)] : true
   <- +available(beer,fridge);
     !has(owner,beer).

// when the fridge is opened, the beer stock is perceived
// and thus the available belief is updated
@a2
+stock(beer,0)
   : available(beer,fridge)
   <- -available(beer,fridge).

@a3
+stock(beer,N)
   : N > 0 & not available(beer,fridge)
   <- +available(beer,fridge).
MAS CONFIGURATION

MAS domestic_robot {
  environment: HouseEnv(gui)
  agents:
    robot;
    owner;
    supermarket agentArchClass SupermarketArch;
}
SOME FURTHER FEATURES: HANDLING PLAN FAILURE

- Handling plan failures is very important when agents are situated in dynamic and non-deterministic environments.
- Plan failure in Jason is represented by goal-deletion event – ! g
  - allows for specifying repairing plans
  - example:

```
+!process_file(F)
  <- !check_existence(F);
    !read_content(F).

-!process_file(F)
  <- !create_file(F);
    ...  
```
SOME FURTHER FEATURES: BELIEF AND PLAN ANNOTATIONS

- Beliefs and plans can be annotated with meta-level information
  - Beliefs: \( \text{bel}\_\text{literal} \ [a_1,a_2,\ldots] \)
    - example
      - \( \text{temperature}(\text{reactor},70) \ [\text{degOfCert}(0.7)] \)
    - pre-defined: \( \text{source}(X), X \in \{\text{self,percept,AgentName}\} \)
      - \( \text{myLocation}(6,5)[\text{source}(\text{self})], \text{red(box1)}[\text{source}(\text{percept})], \text{blue(box1)}[\text{source}(\text{ag1})] \)
  - Plans: \( \text{@plan}\_\text{label} \ [a_1,a_2,\ldots] \)
    - custom selection functions can use such information in plan/intention selection
      
    ```
    @aPlan[ \text{chance}\_\text{of}\_\text{success}(0.3), \text{usual}\_\text{payoff}(0.9)]
    +!g(X) : c(t)
    <- a(X).
    ```

- (chance of success * usual payoff) is the expected utility for that plan
CUSTOMISING THE INFRASTRUCTURE
CUSTOMISING THE INFRASTRUCTURE

• Jason makes it possible to easily customise in Java many aspects of the MAS
  - by extending/specialising Java classes inside the Jason interpreter
CUSTOMISING THE INFRASTRUCTURE

- Jason makes it possible to easily customise in Java many aspects of the MAS
  - by extending/specialising Java classes inside the Jason interpreter
- Among the others:
  - extending the set of internal actions
  - specialising the agent architecture
  - specialising the agent reasoning cycle function
USER-DEFINED INTERNAL ACTIONS

• Programmers can extend the set of internal actions by adding new ones implemented in Java (or other programming languages)
  - collected in modules (libraries), whose name corresponds to the package name

• Example of user defined internal action:

```java
package myLib;
import jason.JasonException;
import jason.asSemantics.*;
import jason.asSyntax.*;

public class randomInt extends DefaultInternalAction {
    private java.util.Random random = new java.util.Random();

    public Object execute(TransitionSystem ts, Unifier un, Term[] args) throws Exception {
        if (!args[0].isNumeric() || !args[1].isVar())
            throw new JasonException("check arguments");
        try {
            int R = random.nextInt( ((numberTerm)args[0]).solve() );
            return un.unifies(args[1], new NumberTermImpl(R));
        } catch (Exception e) {
            throw new JasonException("Error in internal action 'randomInt'", e);
        }
    }
}
```

- can be used inside agent code as `myLib.randomInt(X)`
CUSTOMISING THE AGENT ARCHITECTURE

• Users can define a specific way the agent interacts with the multi-agent systems infrastructure
  - this is used to customise the way the agent does perception of the environment, receives communication massages, and acts in the environment

• Specifying the class in the configuration file:
  - agents: ag1 agentArchClass MyAgArch;

• Example of customised architecture class:

```java
public class MyAgArch extends AgentArchitecture {
    public void perceive() {
        System.out.println("Getting percepts!");
        super.perceive();
    }
}
```
CUSTOMISING THE AGENT REASONING CYCLE
CUSTOMISING THE AGENT REASONING CYCLE

• This is used to customise the selection functions of the AgentSpeak interpreter and other agent-specific functions
  - selection functions
  - belief update and revision
  - functions defining trust/power relations for processing communication messages
  - ...

A. Ricci
Programming Agents and MAS - Scuola Dottorato in Ing. e Scienza dell'Informazione
WRAP-UP
WRAP-UP

• Agent-Oriented programming languages
  - keeping a suitable abstraction level when programming an agent system
  • first-class language support for defining goals, tasks, agent knowledge, plans, etc.
  - focus on individual agent programming + support for agent direct communication
  • based on some high-level ACL (agent communication language)
WRAP-UP

- Agent-Oriented programming languages
  - keeping a suitable abstraction level when programming an agent system
  - first-class language support for defining goals, tasks, agent knowledge, plans, etc.
  - focus on individual agent programming + support for agent direct communication
    - based on some high-level ACL (agent communication language)
- Jason case
  - agent programming language based on the BDI model/architecture
    - equipped with features that make the language useful in practice
    - well-designed and customisable Java-based interpreter/platform
PART 2:
MULTI-AGENT PROGRAMMING
FROM AOP TO MAP
FROM AOP TO MAP

• Agent-Oriented Programming
  - main focus on the individual agent level
FROM AOP TO MAP

- Agent-Oriented Programming
  - main focus on the individual agent level
- Multi-Agent Programming
  - focus shift
    - from the individual-level to the *(multi-agent) system level*
    - interaction and related themes as main dimensions
  - introducing first-class abstractions and mechanisms to help the design and development of systems of agents
FROM AOP TO MAP

• Agent-Oriented Programming
  - main focus on the individual agent level
• Multi-Agent Programming
  - focus shift
    • from the individual-level to the \textit{(multi-agent) system level}
    • \textbf{interaction} and related themes as main dimensions
  - introducing first-class abstractions and mechanisms to help the design and development of systems of agents
• Among the research topics
  - \textit{agent communication languages and protocols}
  - \textit{organisation-oriented programming}
  - \textit{coordination models, languages and infrastructures}
  - \textit{environment programming}
AGENT COMMUNICATION
AGENT COMMUNICATION

- Two approaches (families)
  - **direct** communication
    - Agent Communication Languages
  - **mediated** communication
    - coordination models, languages and infrastructures
      - PART 2.1
    - environment-based
      - PART 2.2
AGENT COMMUNICATION LANGUAGES
AGENT COMMUNICATION LANGUAGES

• Languages developed specifically for agent communication
  - influenced by *speech act* theory (Austin [Au69], Searle [Sea69])
  • treating communication as action
  - conceiving speech acts as *rational* actions
  • performed by agents like other actions, *in the furtherance of their intentions*
AGENT COMMUNICATION LANGUAGES

• Languages developed specifically for agent communication
  - influenced by speech act theory (Austin [Au69], Searle [Sea69])
  • treating communication as action
  - conceiving speech acts as rational actions
  • performed by agents like other actions, in the furtherance of their intentions

• Main examples
  - **KQML** (Knowledge Query and Manipulation Language)
    • message-based language for agent communication
    • first ACL to be developed, in 90ies
  - **FIPA ACL**
    • FIPA (Foundation for Intelligent Physical Agents) initiative
    • improving KQML towards a standard ACL with a well-defined semantics
A TASTE OF KQML

• KQML defines a common format for messages, which includes:
  - a **performative**
    • defines the type of speech acts, the goal of the act
      • tell, ask-one, achieve, advertise
  - content
    • to be described with other languages
      • an example is KIF = Knowledge Interchange Format
  - the receiver, the sender, the ontology, .. and other attributes that depend on the specific performative

• Example

```prolog
(ask-one
 :content (PRICE IBM ?price)
 :receiver stock-server-agent
 :language LPROLOG
 :ontology NYSE-TICKS )
```
FIPA ACL AND JADE PLATFORM
FIPA ACL AND JADE PLATFORM

- FIPA organisation’s aim
  - defining standards for enabling interoperability among agents written using heterogeneous platforms/technologies
  - deliverables: *FIPA specification*
- FIPA ACL is the core component
FIPA ACL AND JADE PLATFORM

• FIPA organisation’s aim
  - defining standards for enabling interoperability among agents written using heterogeneous platforms/technologies
  - deliverables: *FIPA specification*
    • FIPA ACL is the core component

• A FIPA Compliant Platform: **JADE** (Java Agent DEvelopment) Framework [BCG07]
  - one of the most used Java-based platform to develop agent systems
    • in the industry, in particular
  - middleware compliant with FIPA specification
    • implementing a rich support for FIPA ACL and agent communication protocols
FROM DIRECT
TO MEDIATED COMMUNICATION
FROM DIRECT TO MEDIATED COMMUNICATION

- ACL promotes *direct* communication among agents
  - point-to-point communication
  - abstracting from the *medium* that enable that communication
FROM DIRECT TO MEDIATED COMMUNICATION

• ACL promotes direct communication among agents
  - point-to-point communication
  - abstracting from the medium that enable that communication

• Direct communication is not always the best solution when we need to design and manage complex interaction inside a multi-agent system
  - human society example
    • designing proper tool to support human coordination besides verbal point-to-point communication
    • many examples: blackboards, forums, blogs, etc
FROM DIRECT TO MEDIATED COMMUNICATION

- ACL promotes *direct* communication among agents
  - point-to-point communication
  - abstracting from the *medium* that enable that communication
- Direct communication is not always the best solution when we need to design and manage complex interaction inside a multi-agent system
  - human society example
    - designing proper *tool* to support human coordination besides verbal point-to-point communication
    - many examples: *blackboards, forums, blogs, etc*
- Introducing *mediated* communication
  - focusing on communication and coordination media or tools as first-class abstractions in MAS design and development
PART 2.1:
COORDINATION MODELS, LANGUAGES
AND TECHNOLOGIES
(E. DENTI)
PART 2.2:
ENVIRONMENT PROGRAMMING
BACK TO THE AGENT DEFINITION...
BACK TO THE AGENT DEFINITION...

• The notion of environment is intrinsically related to the notion of agent and multi-agent system
BACK TO THE AGENT DEFINITION...

- 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 and Jennings, [WJ95])
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 and Jennings, [WJ95])

- “An agent is anything that can be viewed as perceiving its environment through sensors and acting upon the environment through effectors.” (Russel and Norvig, [RN03])
BACK TO THE AGENT DEFINITION...

- 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 and Jennings, [WJ95])
  - "An agent is anything that can be viewed as perceiving its environment through sensors and acting upon the environment through effectors." (Russel and Norvig, [RN03])

- Including both physical and software environments
SINGLE-AGENT PERSPECTIVE

(Wooldridge [W92])
• Perception
  - process inside agent inside of attaining awareness or understanding sensory information, creating **percepts**
  • perceived form of external stimuli or their absence
SINGLE-AGENT PERSPECTIVE

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

- **Actions**
  - the means to affect (change or inspect) the environment

(Wooldridge [W92])
MULTI-AGENT PERSPECTIVE

(Jennings, [Jen01])
MULTI-AGENT PERSPECTIVE

- In evidence (Jennings, [Jen01])
  - spheres of visibility and influence
  - overlapping => interaction
WHY ENVIRONMENT PROGRAMMING
WHY ENVIRONMENT PROGRAMMING

- Basic level
  - to define actions, perceptions, and related data-model
  - e.g. to test the MAS
WHY ENVIRONMENT PROGRAMMING

• Basic level
  - to define actions, perceptions, and related data-model
    • e.g. to test the MAS

• Advanced level
  - to uniformly encapsulate and modularise functionalities of the MAS out of the agents, into the environment
    • interaction, coordination, organisation, security,…
  - changing the perspective on the environment
    • environment as a first-class abstraction of the MAS [WOO07]
    • programmable environments [RPV10,RPVO09]
ENVIRONMENT AS FIRST-CLASS ABSTRACTION
ENVIRONMENT AS FIRST-CLASS ABSTRACTION

- Three supporting levels [WOO07]
  - basic level
  - abstraction level
  - interaction-mediation level
BASIC LEVEL

• The environment enables agents to access the deployment context, i.e. the hardware and software and external resources with which the MAS interacts
  • sensors and actuators, a printer, a network, a database, a Web service, etc.

(Figure from [WOO07])
ABSTRACTION LEVEL

- **Bridges the conceptual gap** between the agent abstraction and low-level details of the deployment context
  - shields low-level details of the deployment context

(Figure from [WOO07])
INTERACTION-MEDIATION LEVEL

- **Regulate** the access to shared resources
- **Mediate interaction** between agents

(Figure from [WOO07])
ENVIRONMENT DEFINITION REVISED

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.
RESPONSIBILITIES
RESPONSIBILITIES

• Structuring the MAS
  - the environment is first of all a shared "space" for the agents, resources, and services which structures the whole system
RESPONSIBILITIES

- Structuring the MAS
  - the environment is first of all a shared “space” for the agents, resources, and services which structures the whole system

- Kind of structuring
  - physical structure
    - refers to spatial structure, topology, and possibly distribution, see e.g.,
  - communication structure
    - refers to infrastructure for message transfer, infrastructure for stigmergy, or support for implicit communication
  - social structure
    - refers to the organizational structure of the environment in terms of roles, groups, societies
RESPONSIBILITIES
RESPONSIBILITIES

- Embedding **resources** and **services**
  - resources and services can be situated either in the physical structure or in the abstraction layer introduced by the environment
  - the environment should provide support at the abstraction level shielding low-level details of resources and services to the agents
RESPONSIBILITIES

• Embedding resources and services
  - resources and services can be situated either in the physical structure or in the abstraction layer introduced by the environment
  - the environment should provide support at the abstraction level shielding low-level details of resources and services to the agents

• Encapsulating a state and processes
  - besides the activity of the agents, the environment can have processes of its own, independent of agents
    • example: evaporation, aggregation, and diffusion of digital pheromones
  - It may also provide support for maintaining agent-related state
    • for example, the normative state of an electronic institution or tags for reputation mechanisms
RESPONSIBILITIES

• **Ruling** and **mediating** function
  - the environment can define different types of rules on all entities in the MAS.
  - constraints imposed by the domain at hand or laws imposed by the designer
  - may restrict the access of specific resources or services to particular types of agents, or determine the outcome of agent interactions.
  - preserving the agent system in a consistent state according to the properties and requirements of the application domain
RESPONSIBILITIES

- **Ruling** and **mediating** function
  - the environment can define different types of rules on all entities in the MAS.
  - constraints imposed by the domain at hand or laws imposed by the designer
  - may restrict the access of specific resources or services to particular types of agents, or determine the outcome of agent interactions.
  - preserving the agent system in a consistent state according to the properties and requirements of the application domain

- **Examples**
  - coordination infrastructures
  - e-Institutions
ENVIRONMENT PROGRAMMING
ENVIRONMENT PROGRAMMING

- Environment in the loop of MAS design and programming (Ricci, Piunti, Viroli [RVP10])
  - environment as first-class design & programming abstraction
    - software designers and engineers perspective
    - programming MAS = programming Agents + programming Environment
  - environment as first-class runtime abstraction for agents
    - agent perspective
    - to be observed, used, adapted, constructed, ...
ENVIRONMENT PROGRAMMING

- Environment in the loop of MAS design and programming (Ricci, Piunti, Viroli [RVP10])
  - environment as first-class design & programming abstraction
    - software designers and engineers perspective
    - programming MAS = programming Agents + programming Environment
  - environment as first-class runtime abstraction for agents
    - agent perspective
    - to be observed, used, adapted, constructed, ...
- Defining computational and programming models also for the environment part
PROGRAMMING MODEL: DESIDERATA (1/2)
PROGRAMMING MODEL:
DESIDERATA (1/2)

• Abstraction
  - keeping the agent abstraction level
  • e.g. no agents sharing and calling OO objects
  - effective programming models
  • for controllable and observable computational entities
PROGRAMMING MODEL: DESIDERATA (1/2)

- **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
PROGRAMMING MODEL: DESIDERATA (1/2)

• **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 MODEL:
DESIDERATA (2/2)
PROGRAMMING MODEL: DESIDERATA (2/2)

• (Dynamic) extendibility
  - dynamic construction, replacement, extension of environment parts
  - support for open systems
PROGRAMMING MODEL:
DESIDERATA (2/2)

• (Dynamic) extendibility
  - dynamic construction, replacement, extension of environment parts
  - support for open systems

• Reusability
  - reuse of environment parts in different application contexts / domains
AGENTS AND ARTIFACTS (A&A): BACKGROUND METAPHOR

A. Ricci

Programming Agents and MAS - Scuola Dottorato in Ing. e Scienza dell’Informazione
THE NOTION OF ENVIRONMENT IN A&A

• Agents & Artifacts (A&A) metamodel [ORV08,RVO07]
THE NOTION OF ENVIRONMENT IN A&A

- **Agents & Artifacts** (A&A) metamodel [ORV08,RVO07]
- Environment as a shared computational world...
  - ...modularised in terms of artifacts
  - ...that agents can **use** but also dynamically **adapt / extend** for their individual and collective goals
THE NOTION OF ENVIRONMENT IN A&A

- **Agents & Artifacts** (A&A) metamodel [ORV08,RVO07]
- Environment as a shared computational world...
  - ...modularised in terms of artifacts
  - ...that agents can use but also dynamically adapt / extend for their individual and collective goals
- Twofold perspective
  - first-class abstraction for sw engineers
    - • design, programming perspective
  - first-class entity for agents
    - • runtime perspective
A&A BASIC CONCEPTS
A&A BASIC CONCEPTS

- **Agents**
  - autonomous, goal-oriented pro-active entities
  - create and co-use artifacts for supporting their activities
- besides direct communication
A&A BASIC CONCEPTS

- **Agents**
  - autonomous, goal-oriented pro-active entities
  - create and co-use artifacts for supporting their activities
    - besides direct communication

- **Artifacts**
  - *non-autonomous, function*-oriented, steteful entities
    - controllable and observable
  - modelling the tools and resources used by agents
    - designed by MAS programmers
A&A BASIC CONCEPTS

• **Agents**
  - autonomous, goal-oriented pro-active entities
  - create and co-use artifacts for supporting their activities
    • besides direct communication

• **Artifacts**
  - *non-autonomous, function*-oriented, steteful 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 META-MODEL OVERVIEW

- Manual
- Workspace
- Environment
- Artifact
- Operation
- Observable Event
- Observable Property
- Agent

- Has
- Link
- Update
- Generate
- Join
- Quit
- Consult
- Create dispose link
- Use
- Perceive
- Observe
- Perceive

A. Ricci
Programming Agents and MAS - Scuola Dottorato in Ing. e Scienza dell’Informazione
“Coffee-machine” metaphor
EXAMPLES

- A counter
  - count 5
  - inc
  - reset

- A flag
  - state true
  - switch

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

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

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

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

- An event service
  - clearEvents
  - postEvent
  - registerForEvs

- A tuple space
  - out
  - in
  - rd
CATEGORIES OF ARTIFACTS
CATEGORIES OF ARTIFACTS

• **Personal** artifacts
  - designed to provide functionalities for a single agent use
  • examples: agenda,...
CATEGORIES OF ARTIFACTS

• *Personal* artifacts
  - designed to provide functionalities for a single agent use
  • examples: agenda,...

• *Social* artifacts
  - designed to provide some kind of global functionalities
  • communication, coordination, organisation...
  • examples: blackboards, tuple spaces, bounded buffers, ...
CATEGORIES OF ARTIFACTS

• **Personal artifacts**
  - designed to provide functionalities for a single agent use
  • examples: agenda,...

• **Social artifacts**
  - designed to provide some kind of global functionalities
  • communication, coordination, organisation...
  • examples: blackboards, tuple spaces, bounded buffers, ...

• **Boundary artifacts**
  - designed to wrap the interaction with external systems or to represent inside the MAS an external system
  • examples: data-base, GUI, Web Services, ...
• Abstraction
  - artifacts as first-class resources and tools for agents
ABSTRACTION & MODULARIZATION

• Abstraction
  - artifacts as first-class resources and tools for agents

• Modularization
  - artifacts as dynamic reusable modules encapsulating functionalities, organized in workspaces
  - distribution
  - “Divide and Conquer”
ABSTRACTION LAYER (1/2)

Using artifacts to represent or interact with external resources / users.
ABSTRACTION LAYER (2/2)

Using artifacts to provide *internal* resources enabling some kind of functionalities to agents

- e.g. coordinating functionalities, such as blackboard
- e.g. a personal agenda, enhancing agent capabilities
AGENT-ARTIFACT INTERACTION

- Defining actions and percepts in artifact-based environments
  - actions $\leftrightarrow$ artifacts’ operation
    - the action repertoire is given by the set of operations provided by the overall set of artifacts available in the workspace
      - dynamic, can be changed by creating/disposing artifacts
    - action success/failure semantics is defined by operation semantics
  - percepts $\leftrightarrow$ artifacts’ observable properties + signals
    - properties represent percepts about the state of the environment
    - signals represent percepts concerning events signaled by the environment
INTERACTION MODEL: USE

Agent

op(Parms)
INTERACTION MODEL: USE

- Performing an action corresponds to executing an artifact's operation
  - acting on artifact's usage interface
INTERACTION MODEL: USE

OBS PROPERTIES CHANGE

SIGNALS

AGENT

op(params) action

op(Params)

operation execution

action completion - with success or failure -

PropName

Value

Value

...
INTERACTION MODEL: USE

- artifact 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
INTERACTION MODEL: USE

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

**Belief base (or alike)**

**PropName(Value). PropName(Value). ...**

**Agent**

**Observer**

**Focus**
agents can dynamically select which artifacts to observe
-  focus/stopFocus actions
INTERACTION MODEL: OBSERVATION

Belief base (or alike)

PropName(Value).
PropName(Value).
...

AGENT OBSERVER

use
• 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 COMPOSITION / INTERACTION

- Basic mechanism to enable inter-artifact interaction
  - “linking” artifacts through interfaces (link interfaces)
- Operations triggered by an artifact over another artifact

Useful to design & program distributed environments
- realised by set of artifacts linked together
- possibly hosted in different workspaces
ARTIFACT MANUAL
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
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
  - dynamically choosing which artifacts to use to accomplish their tasks and how to use them
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
  - dynamically choosing which artifacts to use to accomplish their tasks and how to use them
- Strong links with Semantic Web research issues
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
  - dynamically choosing which artifacts to use to accomplish their tasks and how to use them
• Strong links with Semantic Web research issues
• Work in progress
  - defining ontologies and languages for describing the manuals
CArtAgO
CArtAgO

- Computational framework / infrastructure to implement and run artifact-based environment [RPV10, RPVO09, RVO07b]
  - Java-based programming model for defining artifacts
  - set of basic API for agent platforms to work within artifact-based environment
- integration with heterogeneous agent programming platforms
CArtAgO

- Computational framework / infrastructure to implement and run artifact-based environment [RPV10, RPVO09, RVO07b]
  - Java-based programming model for defining artifacts
  - set of basic API for agent platforms to work within artifact-based environment
    - integration with heterogeneous agent programming platforms
- 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
CArtAgO

- Computational framework / infrastructure to implement and run artifact-based environment [RPV10,RPVO09,RVO07b]
  - Java-based programming model for defining artifacts
  - set of basic API for agent platforms to work within artifact-based environment
    - integration with heterogeneous agent programming platforms
- 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 AGENT LANGUAGES AND PLATFORM
INTEGRATION WITH AGENT LANGUAGES AND PLATFORM

- Integration with existing agent platforms [RPABHD08]
  - available bridges: Jason, Jadex, AgentFactory, simpA
- ongoing: 2APL
INTEGRATION WITH AGENT LANGUAGES AND PLATFORM

• Integration with existing agent platforms [RPABHD08]
  - available bridges: Jason, Jadex, AgentFactory, simpA
  • ongoing: 2APL

• 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 data-model based on OOP
JaCa PLATFORM
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 (possibly remote) workspaces
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 (possibly remote) workspaces

• Mapping
  - actions
    • Jason agent external actions are mapped onto artifacts’ operations
  - percepts
    • artifact’s observable properties are mapped onto agent beliefs
    • artifact’s 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

count 5

```java
public class Counter extends Artifact {
    void init() {
        defineObsProperty("count", 0);
    }

    @OPERATION void inc() {
        ObsProperty prop = getObsProperty("count");
        prop.updateValue(prop.intValue() + );
        signal("tick");
    }
}
```
EXAMPLE #1: A SIMPLE COUNTER ARTIFACT

```java
public class Counter extends Artifact {
    void init(){
        defineObsProperty("count", 0);
    }

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

- Some API glances
  - Artifact base class
  - @OPERATION annotation to mark artifact’s operations
  - primitives to work define/update/.. observable properties
  - primitives to generate signals
EXAMPLE #1: USERS AND OBSERVERS AGENTS

- Working with the shared counter

**USER(S)**

```
!create_and_use.

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

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

**OBSERVER(S)**

```
!observe.

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

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

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

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

-?myTool(CounterId): true
  <- .wait(10);
  ?myTool(CounterId).
```
PRE-DEFINED ARTIFACTS
PRE-DEFINED ARTIFACTS

- Each workspace by default contains a predefined set of artifacts providing core and auxiliary functionalities
PRE-DEFINED ARTIFACTS

• Each workspace by default contains a predefined set of artifacts providing core and auxiliary functionalities

• Among the others
  - "workspace", type: cartago.WorkspaceArtifact
    • core 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,...  
  - "ts", type cartago.tools.TupleSpace
    • operations: out, in, rd, ...
  - ....
EXAMPLE #2: A BOUNDED-BUFFER

- **bounded-buffer** artifact for producers-consumers scenarios
  - output operation parameters to represent action feedbacks
  - operation with guards

```java
public class BoundedBuffer extends Artifact {

    private LinkedList<Object> items;
    private int nmax;

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

    @OPERATION(guard="bufferNotFull") void put(Object obj){
        items.add(obj);
        getObsProperty("n_items").updateValue(items.size());
    }

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

    @GUARD boolean itemAvailable(OpFeedbackParam<Object> res){
        return items.size() > 0;
    }

    @GUARD boolean bufferNotFull(Object obj){
        return items.size() < nmax;
    }
}
```
**PRODUCERS & CONSUMERS AGENTS**

### PRODUCERS

```prolog
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

```prolog
!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.
```
EXAMPLE #3: A TUPLE SPACE

- A simple tuple space implemented as an artifact
  - operations with multiple transactional guarded step

```java
public class TupleSpace extends Artifact {
    TupleSet tset;

    void init(){
        tset = new TupleSet();
    }

    @OPERATION void out(String name, Object... args){
        tset.add(new Tuple(name, args));
    }

    @OPERATION void in(String name, Object... params){
        TupleTemplate tt = new TupleTemplate(name, params);
        await("foundMatch", tt);
        Tuple t = tset.removeMatching(tt);
        bind(tt, t);
    }

    @OPERATION void rd(String name, Object... params){
        TupleTemplate tt = new TupleTemplate(name, params);
        await("foundMatch", tt);
        Tuple t = tset.readMatching(tt);
        bind(tt, t);
    }

    private void bind(TupleTemplate tt, Tuple t){ ... }

    @GUARD boolean foundMatch(TupleTemplate tt){
        return tset.hasTupleMatching(tt);
    }
}
```
DINING PHILOSOPHER AGENTS
EXPLOITING A TUPLE SPACE ARTIFACT

WAITER

philo(0,"philo1",0,1).
philo(1,"philo2",1,2).
philo(2,"philo3",2,3).
philo(3,"philo4",3,4).
philo(4,"philo5",4,0).

!prepare_table.

!prepare_table
<= for (.range(I,0,4)) {
  out("fork",I);
  ?philo(I,Name,Left,Right);
};
for (.range(I,1,4)) {
  out("ticket");
};
println("done.").

PHILO

!start.

!start
<= .my_name(Me);
in("philo_init",Me,Left,Right);
+my_left_fork(Left);
+my_right_fork(Right);
println(Me," ready.");
!!living.

!living
<= .thinking; !eating; !!living.

!eating
<= !acquireRes; !eat; !releaseRes.

!acquireRes : my_left_fork(F1) & my_right_fork(F2)
<= in("ticket"); in("fork",F1); in("fork",F2).

!releaseRes: my_left_fork(F1) & my_right_fork(F2)
<= out("fork",F1); out("fork",F2); out("ticket").

!thinking <= .my_name(Me); println(Me," thinking").
!eat <= .my_name(Me); println(Me," eating").
EXAMPLE #4: A CLOCK

- Clock artifact
  - execution of operations triggered by other operations

public class Clock extends Artifact {
    boolean working;
    final static long TICK_TIME = 100;

    void init(){
        working = false;
    }

    @OPERATION void start(){
        if (!working){
            working = true;
            execInternalOp("work");
        } else {
            failed("already_working");
        }
    }

    @OPERATION void stop(){
        working = false;
    }

    @INTERNAL_OPERATION void work(){
        while (working){
            signal("tick");
            await_time(TICK_TIME);
        }
    }
}

// clock user

!test_clock.

+!test_clock
    <- makeArtifac("myClock","Clock",[],Id);
    focus(Id);
    +n_ticks(0);
    start;
    println("clock started.").

@plan1
+tick: n_ticks(10)
    <- stop;
    println("clock stopped.").

@plan2 [atomic]
+tick: n_ticks(N)
    <- -+n_ticks(N+1);
    println("tick perceived!").
EXAMPLE #5: GUI ARTIFACTS

- Exploiting artifacts to enable interaction between human users and agents.
EXAMPLE #5: GUI ARTIFACTS

```java
import javax.swing.*;
import java.awt.event.*;
import cartago.*;
import cartago.tools.*;

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");
    }

    @INTERNAL_OPERATION void closed(WindowEvent ev){
        signal("closed");
    }

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

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

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

// gui tester agent.
!test_gui.

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

+value(V)

+ok : value(V)
    <- setValue(V+1).

+closed
    <- .my_name(Me);
        .kill_agent(Me).
```

```java
class MyFrame extends JFrame {
    private JButton okButton;
    private JTextField text;

    public MyFrame(){
        setTitle("Simple GUI ");
        setSize(200,100);
        JPanel panel = new JPanel();
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); // gui tester agent.
        okButton = new JButton("ok");
        okButton.setSize(80,50);
        text = new JTextField(10);
        text.setText("0");
        panel.add(text);
        panel.add(okButton);
}

    public String getText(){ return text.getText(); }
    public void setText(String s){ text.setText(s); }
}
```
CArtAgO AND JaCa:
SOME REAL-WORLD APPLICATIONS
CArtAgO AND JaCa: SOME REAL-WORLD APPLICATIONS

- JaCa-WS / CArtAgO-WS
  - building SOA/Web Services applications using JaCa
CArtAgO AND JaCa:  
SOME REAL-WORLD APPLICATIONS  

- **JaCa-WS / CArtAgO-WS**  
  - building SOA/Web Services applications using JaCa  
  - [http://cartagows.sourceforge.net](http://cartagows.sourceforge.net)  

- **JaCa-Web**  
  - implementing Web 2.0 applications using JaCa  
CArtAgO AND JaCa: SOME REAL-WORLD APPLICATIONS

- **JaCa-WS / CArtAgO-WS**
  - building SOA/Web Services applications using JaCa
  - [http://cartagows.sourceforge.net](http://cartagows.sourceforge.net)

- **JaCa-Web**
  - implementing Web 2.0 applications using JaCa

- **JaCa-Android**
  - implementing mobile computing applications on top of the Android platform using JaCa
WRAP-UP

- Environment programming
  - environment as a programmable part of the MAS
  - programming agents’ world for agents’ use
WRAP-UP

- Environment programming
  - environment as a programmable part of the MAS
  - programming agents’ world for agents’ use
- Artifact-based computational & programming model
  - 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
WRAP-UP

• Environment programming
  - environment as a programmable part of the MAS
  - programming agents’ world for agents’ use

• Artifact-based computational & programming model
  - 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
  - a computational framework for programming and executing artifact-based environments
  - integration with heterogeneous agent platform

• JaCa case
REFERENCES


REFERENCES


REFERENCES


