Programming Intentional Agents in AgentSpeak(L) and Jason

Multiagent Systems LS
Sistemi Multiagente LS

Andrea Omicini & Michele Piunti
{andrea.omicini, michele.piunti}@unibo.it

Ingegneria Due
Alma Mater Studiorum—Università di Bologna a Cesena

Academic Year 2009/2010
1 Implementing BDI Architectures

2 AgentSpeak(L)
   - Syntax
   - Semantics

3 Jason
   - Reasoning Cycle
   - Jason Programming Language
   - Jason Agents Playing with CArtAgO working environments
   - Advanced BDI aspects

4 Conclusions and References
BDI Abstract Control Loop [Rao and Georgeff, 1995]

1. initialize-state();
2. while true do
3.   options := option-generator(event-queue);
4.   selected-options := deliberate(options);
5.   update-intentions(selected-options);
6.   execute();
7.   get-new-external-events();
8.   drop-successful-attitudes();
9.   drop-impossible-attitudes();
10. end-while
Structure of BDI Systems

BDI architectures are based on the following constructs

1. **A set of beliefs**
2. **A set of desires (or goals)**
3. **A set of intentions**
   - Or better, a subset of the goals with an associated stack of plans for achieving them. These are the intended actions;
4. **A set of internal events**
   - Elicited by a belief change (i.e., updates, addition, deletion) or by goal events (i.e., a goal achievement, or a new goal adoption).
5. **A set of external events**
   - Perceptive events coming from the interaction with external entities (i.e., message arrival, signals, etc.)
6. **A plan library (repertoire of actions)** as a further (static) component.
Basic Architecture of a BDI Agent [Wooldridge, 2002]

Implementing BDI Architectures

Agent

- Sensors
  - BRF
    - Beliefs
      - Generate Options
        - Generate Options
          - Desires
            - Intentions
              - Action

- Effectors
  - Filter

Omicini & Piunti (Università di Bologna) Programming Intentional Agents A.Y. 2009/2010
Procedural Reasoning System (PRS)  
[Georgeff and Lansky, 1987]

- PRS is one of the first BDI architectures (developed by M.P. Georgeff and A.L. Lansky)
- PRS is a goal directed and reactive planning system
- Goal directedness allows reasoning about and performing complex tasks
- Reactiveness allows handling real-time behaviour in dynamic environments
- PRS is applied for high-level reasoning of robot, airport traffic control systems etc.
PRS Architecture

- Data Input
- Data Base (Beliefs)
- Goals (Desires)
- Stack (Intentions)
- Command Generator
- Monitor
- KAS (Plans)
- Effectors
- Sensors
- System Interface
- Environment

Interpreter (Reasoner)
AgentSpeak(L) is an abstract language used for describing and programming BDI agents.

Inspired by PRS, dMARS (Kinny), and BDI Logics (Rao and Georgeff).

Originally proposed by Anand S. Rao [Rao, 1996]

AgentSpeak(L) is extended to make it a practical agent programming language [Bordini and Hübner, 2006]

AgentSpeak(L) programs can be executed by the Jason platform [Bordini et al., 2007]

Operational semantics for extensions of AgentSpeak(L) which provides a computational semantics for BDI concepts.
Outline

1. Implementing BDI Architectures

2. AgentSpeak(L)
   - Syntax
   - Semantics

3. Jason
   - Reasoning Cycle
   - Jason Programming Language
   - Jason Agents Playing with CArtAgO working environments
   - Advanced BDI aspects

4. Conclusions and References
Syntax of AgentSpeak(L)

- The main language constructs of AgentSpeak are:
  - **Beliefs**: current state of the agent, information about environment, and other agents
  - **Goals**: state the agent desire to achieve and about which he brings about (Practical Reasoning) based on internal and external stimuli
  - **Plans**: recipes of procedural means the agent has to change the world and achieve his goals

- The architecture of an AgentSpeak agent has four main components:
  1. Belief Base
  2. Plan Library
  3. Set of Events
  4. Set of Intentions
Beliefs and Goals

Beliefs

If $b$ is a predicate symbol, and $t_1, ..., t_n$ are (first-order) terms, $b(t_1, ..., t_n)$ is a belief atom

- Ground belief atoms are base beliefs
- If $\Phi$ is a belief atom, $\Phi$ and $\neg\Phi$ are belief literals

Goals

If $g$ is a predicate symbol, and $t_1, ..., t_n$ are terms, $!g(t_1, ..., t_n)$ and $?g(t_1, ..., t_n)$ are goals

1. ‘!’ means Achievement Goals (Goal to do)
2. ‘?’ means Test Goals (Goal to know)
## Beliefs and Goals

### Beliefs

If $b$ is a predicate symbol, and $t_1, \ldots, t_n$ are (first-order) terms, $b(t_1, \ldots, t_n)$ is a belief atom.

- Ground belief atoms are base beliefs.
- If $\Phi$ is a belief atom, $\Phi$ and $\neg \Phi$ are belief literals.

### Goals

If $g$ is a predicate symbol, and $t_1, \ldots, t_n$ are terms, $!g(t_1, \ldots, t_n)$ and $?g(t_1, \ldots, t_n)$ are goals.

1. ‘!’ means Achievement Goals (Goal to do).
2. ‘?’ means Test Goals (Goal to know).
Events

- Events are signalled as a consequence of changes in the agent’s belief base or goal states.
- Events may signal to the agent that some situation is requiring servicing (triggering events).
- The agent indeed is supposed to react to such events by finding a suitable plan(s).
- Due to events and goal processing, AgentSpeak(L) architectures are both
  - Reactive
  - Proactive
Events

If \( b(t) \) is a belief atom, \(!g(t)\) and \(?g(t)\) are goals, then \(+b(t), −b(t), +!g(t), +？g(t), −!g(t), \) and \(−?g(t)\) are triggering events.

Let \( \Phi \) be a literal, then the AgentSpeak triggering events are the following:

- \(+\Phi\) Belief addition
- \(−\Phi\) Belief deletion
- \(+!\Phi\) Achievement-goal addition
- \(−!\Phi\) Achievement-goal deletion
- \(+?\Phi\) Test-goal addition
- \(−?\Phi\) Test-goal deletion
Plans... are recipes for achieving goals... declaratively define a workflow of actions... along with the triggering and the context conditions that must hold in order to initiate the execution... Plans represent agent’s means to achieve goals (their know-how)

<table>
<thead>
<tr>
<th>Plans</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plans If ( e ) is a triggering event, ( b_1, ..., b_n ) are belief literals (plan context), and ( h_1, ..., h_n ) are goals or actions (plan body), then ( e : b_1 \land ... \land b_n \leftarrow h_1; ...; h_n ) is a plan (where ( e : c ) is called the plan’s head)</td>
</tr>
</tbody>
</table>
Plans...

- ... are recipes for achieving goals
- ... declaratively define a workflow of actions
- ... along with the triggering and the context conditions that must hold in order to initiate the execution
- Plans represent agent’s means to achieve goals (their know-how)

**Plans**

If $e$ is a triggering event, $b_1, \ldots, b_n$ are belief literals (plan context), and $h_1, \ldots, h_n$ are goals or actions (plan body), then

$$e : b_1 \land \ldots \land b_n \leftarrow h_1; \ldots; h_n$$

is a plan (where $e : c$ is called the plan’s head)
Plans

Let $\Phi$ be a literal, then the PlanBody (i.e. intentions in AgentSpeak) can include the following elements:

- $!\Phi$ Achievement goals
- $?\Phi$ Test goals
- $+\Phi$ Belief addition
- $-\Phi$ Belief deletion
- $\Phi$ Actions
- $.\Phi$ Internal Actions (*not actually here, this is Jason…*)
An AgentSpeak plan has the following general structure:

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

where:

- The **triggering event** denotes the events that the plan is meant to handle.
- The **context** represents the circumstances in which the plan can be used.
  - logical expression, typically a conjunction of literals to be checked whether they follow from the current state of the belief base (Belief Formulae).
- 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.
  - A sequence of actions and (sub) goals to achieve that goal.
AgentSpeak(L) Examples

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

/* Belief addition */
+concert(Artist, Date, Venue)
  : likes(Artist)
  <- !book_tickets(Artist, Date, Venue).

/* Plan to 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).
Outline

1. Implementing BDI Architectures

2. AgentSpeak(L)
   - Syntax
   - Semantics

3. Jason
   - Reasoning Cycle
   - Jason Programming Language
   - Jason Agents Playing with CArtAgO working environments
   - Advanced BDI aspects

4. Conclusions and References
AgentSpeak(L) Semantics

AgentSpeak(L) has an operational semantics defined in terms of agent configuration $\langle B, P, E, A, I, S_e, S_o, S_I \rangle$
where

- $B$ is a set of beliefs
- $P$ is a set of plans
- $E$ is a set of events (external and internal)
- $A$ is a set of actions that can be performed in the environment
- $I$ is a set of intentions each of which is a stack of partially instantiated plans
- $S_e, S_o, S_I$ are selection functions for events, options, and intentions
The selection functions

- $S_e$ selects an event from $E$. The set of events is generated either by requests from users, from observing the environment, or by executing an intention.

- $S_o$ selects an option from $P$ for a given event. An option is an applicable plan for an event, i.e. a plan whose triggering event is unifiable with event and whose condition is derivable from the belief base.

- $S_I$ selects an intention from $I$ to execute.
Semantics of Intention Execution

- $tr : ct \leftarrow +\varphi; \ldots \Rightarrow$ Generates event $+\varphi$ and updates beliefs. If no applicable plan for $+\varphi$, discard the event.

- $tr : ct \leftarrow -\varphi; \ldots \Rightarrow$ Generates event $-\varphi$ and updates beliefs. If no applicable plan for $-\varphi$, discard the event.

- $tr : ct \leftarrow !\varphi; \ldots \Rightarrow$ Generates event $+!\varphi$. If no applicable plan for $+!\varphi$, remove plan and generate $-!\psi$ if $tr = +!\psi$ (or $-?\psi$ if $tr = +?\psi$).

- $tr : ct \leftarrow ?\varphi; \ldots \Rightarrow$ Generates event $+?\varphi$ If no applicable plan for $+?\varphi$, remove plan and generate $-!\psi$ if $tr = +!\psi$ (or $-?\psi$ if $tr = +?\psi$).

- $tr : ct \leftarrow \varphi; \ldots \Rightarrow$ If the action fails, remove plan and generate $-!\psi$ if $tr = +!\psi$ (or $-?\psi$ if $tr = +?\psi$).

- $tr : ct \leftarrow .\varphi; \ldots \Rightarrow$ If the internal action fails, remove plan and generate $-!\psi$ if $tr = +!\psi$ (or $-?\psi$ if $tr = +?\psi$).

If no plan is applicable for a generated $-!\psi$ or $-?\psi$, then the whole intention is disregarded and an error message is printed.
Agent Configuration

Configuration of an AgentSpeak Agent

\[ \langle ag, C, M, T, s \rangle \]

- \( ag \) is an AgentSpeak program consisting of a set of beliefs and plans
- \( C = \langle I, E, A \rangle \) is the agent circumstance
- \( M = \langle In, Out, SI \rangle \) is the communication component
- \( T = \langle R, Ap, \nu, \varepsilon, \rho \rangle \) is the temporary information component
- \( s \) is the current step within an agent’s reasoning cycle
Circumstance Component

\[ \langle ag, C, M, T, s \rangle \]

Agent's Circumstance

\[ C = \langle I, E, A \rangle \]

- \( I \) is a set of intentions \( \{i, i', \ldots\} \); each intention \( i \) is a stack of partially instantiated plans.

- \( E \) is a set of events \( \{(tr, i), (tr', i'), \ldots\} \); each event is a pair \( (tr, i) \), where \( tr \) is a triggering event and \( i \) is an intention (a stack of plans in case of an internal event or \( T \) representing an external event).

- \( A \) is a set of actions to be performed in the environment; an action expression included in this set tells other architecture components to actually perform the respective action on the environment, thus changing it.
Communication Component

\[ \langle ag, C, M, T, s \rangle \]

Agent’s communication

\[ M = \langle In, Out, SI \rangle \]

- **In** is the mail inbox: the system includes all messages addressed to this agent in this set
- **Out** is where the agent posts all messages it wishes to send to other agents
- **SI** is used to keep track of intentions that were suspended due to the processing of communication messages

Message

\[ \langle messageid, agentid, ilf, content \rangle \]
Communication Component

\[ \langle ag, C, M, T, s \rangle \]

**Agent’s communication**

\[ M = \langle In, Out, SI \rangle \]

- **In** is the mail inbox: the system includes all messages addressed to this agent in this set
- **Out** is where the agent posts all messages it wishes to send to other agents
- **SI** is used to keep track of intentions that were suspended due to the processing of communication messages

**Message**

\[ \langle messageid, agentid, ilf, content \rangle \]
Temporary Information Component

\[ \langle ag, C, M, T, s \rangle \]

**Temporary information**

\[ T = \langle R, Ap, \iota, \varepsilon, \rho \rangle \]

- \( R \) for the set of relevant plans (for the event being handled)
- \( Ap \) for the set of applicable plans (the relevant plans whose context are true)
- \( \iota, \varepsilon \) and \( \rho \) keep record of a particular intention, event and applicable plan (respectively) being considered along the execution of an agent
Deliberation Steps

The current step $s$ within an agent’s reasoning cycle is one of the following elements:

- **ProcMsg**: processing a message from the agent’s mail inbox
- **SelEv**: selecting an event from the set of events
- **RelPl**: retrieving all relevant plans
- **ApplPl**: checking which of those are applicable
- **SelAppl**: selecting one particular applicable plan (the intended means)
- **AddIM**: adding the new intended means to the set of intentions
- **SelInt**: selecting an intention
- **ExecInt**: executing the select intention
- **ClrInt**: clearing an intention or intended means that may have finished in the previous step
Outline

1. Implementing BDI Architectures

2. AgentSpeak(L)
   - Syntax
   - Semantics

3. Jason
   - Reasoning Cycle
   - Jason Programming Language
   - Jason Agents Playing with CArtAgO working environments
   - Advanced BDI aspects

4. Conclusions and References
Developed by Jomi F. Hübner and Rafael H. Bordini

Jason implements the operational semantics of a variant of AgentSpeak [Bordini and Hübner, 2006]

Extends AgentSpeak, which is meant to be the language for defining agents

Adds a set of powerful mechanism to improve agent abilities

Extensions aimed at a more practical programming language
  - High level language to define agents (goal oriented) behaviour
  - Java as low level language to realize mechanisms (i.e. agent internal functions) and customize the architecture

Comes with a framework for developing multi-agent systems

http://jason.sourceforge.net/
Jason Architecture

**Reasoning Cycle**

1. **Percepts**
   - perceive
   - BUF
   - BRF
   - External Events

2. **Belief Base**
   - Beliefs
   - Events
   - Selected Event
   - Relevant Plans
   - New Plan
   - Push

3. **Messages**
   - checkMail
   - S_m

4. **SocAcc**
   - Beliefs to Add and Delete
   - Internal Events

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

6. **Unified Event**
   - Relevant Plans
   - Plans
   - New Intention
   - Intended Means
   - Selected Intention
   - Action

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

8. **S_k**
   -Beliefs
   - S_0
   - Means
   - Intended
   - New Intention

9. **S_0**
   - New Intention
   - Updated Intention
   - SendMsg

10. **Execute Intention**
    - Actions
    - Act
    - Messages

**Agent**

**Suspended Intentions**
- (Actions and Msgs)
- ...
Jason Reasoning Cycle

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
public void reasoningCycle() {
    try {
        C.reset(); // C is actual Circumstance
        if (nrcslbr >= setts.nrcbp()) {
            nrcslbr = 0;
            ag.buf(agArch.perceive());
            agArch.checkMail();
        }
        nrcslbr++;
        // counting number of cycles
        if (canSleep()) {
            if (ag.pl.getIdlePlans() != null) {
                logger.fine("generating idle event");
                C.addExternalEv(PlanLibrary.TE_IDLE);
            } else {
                agArch.sleep();
                return;
            }
        }
        step = State.StartRC;
        do {
            if (!agArch.isRunning()) return;
            applySemanticRule();
        } while (step != State.StartRC);
        ActionExec action = C.getAction();
        if (action != null) {
            C.getPendingActions().put(action.getIntention().getId(), action);
            agArch.act(action, C.getFeedbackActions());
        }
    } catch (Exception e) {
        conf.C.create(); // ERROR in the transition system, creating a new C
    }
}
Outline

1. Implementing BDI Architectures

2. AgentSpeak(L)
   - Syntax
   - Semantics

3. Jason
   - Reasoning Cycle
   - Jason Programming Language
   - Jason Agents Playing with CArtAgO working environments
   - Advanced BDI aspects

4. Conclusions and References
Jason as an Agent Programming Language

- Jason includes all the syntax and the semantics already defined for AgentSpeak
- Boolean operators
  - ==, <, <=, >, >=, &, |, \==, not
- Arithmetic
  - +, -, /, *, **, mod, div
- Then Jason includes several extensions
- For instance: let Φ be a literal, then a Jason PlanBody can include the following additional elements:
  - !!Φ To launch a given plan Φ as a new intention (the new intention will not be related to the current one, its execution will be as if it is in a new thread).
  - –+ Φ To update a Belief Φ in an atomic fashion (atomic deletion and update)
Belief Annotations

Jason introduces the notion of annotated predicates:

\[ p_s(t_1, ..., t_n)[a_1, ..., a_m] \]

where \( a_i \) are first order terms

- All predicates in the belief base have a special annotation \( source(s_i) \)
  where \( s_i \in \{self, percept\} \cup AgId \)
  - myLocation(6,5)[source(self)].
  - red(box1)[source(percept)].
  - blue(box1)[source(ag1)].

- Agent developer can define customised predicates (i.e. grade of certainty on that belief)
  - colourblind(ag1)[source(self),doc(0.7)].
  - liar(ag1)[source(self),doc(0.2)].
Strong Negation

- Strong negation (operator $\sim$) is another Jason extension to AgentSpeak.
- To allow both closed-world and open-world assumptions.

```jason
+!pit_stop(fuel(T), tires(_))
  : not raining & not $\sim$raining /* Lack of knowledge:
    there is no belief indicating raining
    neither belief indicating $\sim$raining */
<- ++$tires(intermediate); /* Atomic Belief Update */
  !fuel(T+2);
...

+!pit_stop(fuel(T), tires(_))
  : raining /* There is a belief indicating raining */
<- ++$tires(rain); /* Atomic Belief Update */
  !fuel(T+5);
...
```

```jason
+!pit_stop(fuel(T), tires(_))
  : $\sim$raining /* There is a belief indicating $\sim$raining */
<- ++$tires(slick); /* Atomic Belief Update */
  !fuel(T);
...
```
Belief Rules

In Jason, beliefs (and their annotations) can be pre-processed with Prolog-like rules:

```prolog
likely_color(Obj, C)
    :- colour(Obj, C)[degOfCert(D1)]
        & not (colour(Obj, _) [degOfCert(D2)]
            & D2 > D1)
        & not ~colour(Obj, B).
```
Handling Plan Failures

Handling plan failures is very important when agents are situated in dynamic and non-deterministic environments.

- Goal-deletion events are another Jason extension to AgentSpeak.

- `-!g`  

To create an agent that is blindly committed to goal `g`:

```jason
+!g(X) : goalstate
    <- true.
+!g(X) : not goalstate
    <- ...
        ?g.
...
-!g : true /* Goal deletion event */
    <- !g.
```
Plan Annotations

Plan can have annotations too (e.g., to specify meta-level information)

- Selection functions (Java) can use such information in plan/intention selection
- Possible to change those annotations dynamically (e.g., to update priorities)
- Annotations go in the plan label

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

- \((\text{chance_of_success} \times \text{usual_payoff})\) is the expected utility for that plan
Internal Actions

- In Jason plans can contain an additional structure: *internal action*. Φ
- Self-Contained actions which code is packed and atomically executed as part of the agent reasoning cycle
- Internal actions can be used for special purpose activities
  - to interact with Java objects
  - to invoke legacy systems elegantly
  - as we will see in the rest of the course, to use *artifacts* in A&A systems
- Example of user defined internal action:
  ```
  userLibrary.userAction(X,Y,R)
  ```
  can be used to manipulate parameters $X$, $Y$ and unify the result of that manipulation in $R$
Defining New Internal Actions

Internal action: myLib.randomInt(M, N) unifies N with a random int between 0 and M.

```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();

    @Override
    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);
        }
    }
}
```
Predefined Internal Actions

- Many internal actions are available for: printing, sorting, list/string operations, manipulating the beliefs/annotations/plan library, waiting/generating events, etc. (see `jason.stdlib`)

- Predefined internal actions have an empty library name

  - `.print(1,X,"bla")` prints out to the console the concatenation of the string representations of the number 1, of the value of variable X, and the string “bla”;
  - `.union(S1,S2,S3)` S3 is the union of the sets S1 and S2 (represented by lists). The result set is sorted;
  - `.desire(D)` checks whether D is a desire: D is a desire either if there is an event with `+!D` as triggering event or it is a goal in one of the agent's intentions;
  - `.intend(I)` checks if I is an intention: I is an intention if there is a triggering event `+!I` in any plan within an intention; just note that intentions can be suspended and appear in E, PA, and PI as well.
  - `.drop_desire(I)` removes events that are goal additions with a literal that unifies with the one given as parameter.
  - `.drop_intention(I)` drops all intentions which would make `.intend` true.
Internal Actions used for Message Passing

**Sender** Agent A sends a message to agent B using a special internal action:

```plaintext
.send(B, ilf, m(X))
.broadcast(ilf, m(X))
```

- B is the unique name of the agent that will receive the message (or a list of names).
- \( ilf \in \{tell, untell, achieve, unachieve, askOne, askAll, askHow, tellHow, untellHow\} \)
- \( m(X) \) the content of the message

**Receiver** Agent B receives the message from A as a triggering event

- Handles it by customizing a reaction:
  ```plaintext
  +m(X)[source(A)] : true
  <- dosomething;...
  ```
To build and deploy a MAS you need to rely on some sort of environment where the agents are situated.

The environment has to be designed (and implemented as well).

There are two ways to do this:

1. Defining perceptions and actions so to operate on specific environments
   - This is done defining in Java lower-level mechanisms, and by specializing the Agent Architecture and Agent classes (see later).

2. Creating a ‘simulated’ environment
   - This is done in Java by extending Jason’s Environment class and using methods such as `addPercept(String Agent, Literal Percept)`.
import jason.*;
import ...;
public class myEnv extends Environment{
    ....
    public myEnv() {
        Literal loc = Literal.parseLiteral("location(3,5)"");
        addPercept(pos1);
    }

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

1. Implementing BDI Architectures
2. AgentSpeak(L)
   - Syntax
   - Semantics
3. Jason
   - Reasoning Cycle
   - Jason Programming Language
   - Jason Agents Playing with CArtAgO working environments
   - Advanced BDI aspects
4. Conclusions and References
C4Jason is an integration technology enabling Jason agents to operate in CArtAgO working environment [Ricci et al., 2008, Ricci et al., 2007]

- Extending repertoire of agent (internal) actions with a basic set to work within artifact-based environments
  - Workspace Management
  - (Artifact) Operation Use
  - Artifact Observation (Observable properties)
  - Artifact instantiation, discovery and management

- Open-source technology
Table: Jason Internal Actions for managing workspaces (1–2), creating, disposing and looking up artifacts (3–5), using artifacts (6–7), and observing artifacts (8–10). Syntax is expressed in a logic-like notation, where italicised items in square brackets are optional.
Agents and Artifact (A&A) Interactions are defined at a language level, through internal actions which realize step by step the workflow of activities required to achieve a given goal (intentional stance).

**Goal Oriented use of Artifacts** Goals can thus be achieved by the mean of operations which have been defined – by the artifact developer – within artifacts control interface. Using artifact operations agents can *externalize* the activities to achieve their goals.

```
cartago.use(ArID,opName(Params),Sensor)
```

**Doxastic use of Artifacts** Artifact Observable properties can be exploited by agents to retrieve strategic information. The mechanism of observation is conceived as a loosely coupled interaction.

```
cartago.observeProperty(ArID,propName,X)
```
Example of Agents in CArtAgO Environments

/* Join a Workspace and Use Artifacts*/
+!join : mySensor(S)
   cartago.joinWorkspace("CartagoBDI","localhost:4010");
   !locate_artifacts;
   ?artifactBel(envBoard, Eid); /* Use the EnvBoard Artifact to enter the playground */
   cartago.use(Eid, enter, S); /* The artifact signals the initial location Loc */
   cartago.sense(S, entered(Loc)); /* Observe the Number of Rooms */
   cartago.observeProperty(Env, NRooms, Nr); /* In EnvBoard this value is an Observable Property*/
   +nRooms(Nr); /* A new belief is added */
   !explore. /* Now it’s time to explore... */

/* Randomly select a Room to explore and commit the intention to explore it*/
+!explore : nRooms(N) & mySensor(S)
   roomsworld.randomInt(N,X);
   +targetRoom(X);
   !go_to_room(X).

/* Retrieve and store artifacts IDs */
+!locate_artifacts
   cartago.lookupArtifact("envBoard", Eid);
   +artifactBel(envBoard, Eid);
   cartago.lookupArtifact("console", IOid);
   +artifactBel(console, IOid).

+!go_to_room(X) <- ...
MAS Configuration

- Jason has a simple language for defining a MAS, where each agent runs on its own AgentSpeak interpreter.
- The environment is provided by a Java class, *that has to be specified*.
- System infrastructure options: *Centralised* or *Saci*.
- Moreover, through mas2j configuration files you can:
  - Specify hosts (i.e. ag1 at host0.unibo.it).
  - Specify the file where agent’s source code is (i.e. agents: ag1 agent1.asl).
  - Indicate the number of instances of an agent (using the same initial beliefs and plan library) (i.e. agents: ag1 #6).
MAS Configuration File Examples

MAS mars {
    infrastructure: Saci
    environment: marsEnv
    agents: r1; r2;
}

/* An A&A Jason-CArtAgO System */
MAS gridWorld {
    environment: alice.c4jason.CEnv

    agents:
    CleanerAg #3 agentArchClass alice.c4jason.CogAgentArch;
    CleanerAg_Artifacts #3 agentArchClass alice.c4jason.Cog2AgentArch;
}
Outline

1. Implementing BDI Architectures

2. AgentSpeak(L)
   - Syntax
   - Semantics

3. Jason
   - Reasoning Cycle
   - Jason Programming Language
   - Jason Agents Playing with CArtAgO working environments
   - Advanced BDI aspects

4. Conclusions and References
Hierarchical Plannig

- Hierarchical abstraction is a well-known principle
- Exhibits a great effectiveness in planning
- Used to reduce a composite intention – or a given task – to a greater number of independent sub-intentions – or sub-tasks – placed at a lower level of abstraction
- An agent can manage at runtime an alternating hierarchy of (meta)goals and plans, which emerge from top-level goals over plans to subgoals and so forth
  - This highly simplifies the structure of plans
  - Allow the plans to be conceived around self-contained actions (the leafs of the goal hierarchy) which can be reused with different purposes too.
Defined having in mind the problem domain (the goal to be achieved) and trying to imagine those fine grained actions which in turn are supposed to accomplish the required activities.

Differently from traditional planning systems, which mainly make an offline planning, Intentional Systems need to plan in dynamic environments and need to cope changing contexts and situations.

Planning Systems is offline. Can create plans to achieve goals by composing actions in repertoire.

BDI planning hybrid approach. The plans are defined at design time and at the language level but their execution is ruled by the architecture (means ends reasoning) according to context conditions (i.e., Jason, Jadex) or planning rules (i.e., 2APL).
An Example: RoomsWorld Scenario

TERMINAL GOAL:
Clean Rooms

Agents have bounded range of sight, i.e., can locate trash items only entering a given room.

To prevent wasting resources (time) agents need a good strategy to coordinate activities.

Each agent should deliberate the room to visit knowing and evaluating the activities performed by others.

- An Artifact-based environment can be conceived with CArtAgO, so as to allow interoperability of agents belonging to heterogeneous platforms.
- Special Artifacts can be designed to ease agent activities, i.e., to provide services (i.e. timing), to enable mediated forms of communication, to coordinate roles (exploiter, explorer).

[Piunti and Ricci, 2009]
Cleaner Agents

Individuate a structure for goals, plans and actions so as to achieve the terminal goal(s)

PLAN HIERARCHY
Individuate a basic set of Beliefs which are mandatory for Means Ends Reasoning

GOAL SUPPORTING BELIEFS

[Piunti and Ricci, 2008]
Goal Supporting Beliefs

- trash_at_room(N)
- trash(Tx, Ty)
- loc(X, Y)
- room(N)
- go_to_room
- clean_trash
- go_to_XY
- move
- epistemic_action
- cleanTrash

Omicini & Piunti (Università di Bologna) Programming Intentional Agents A.Y. 2009/2010 58 / 71
Cleaner Agents in Jason

/* A message event from an explorer agent is signalling the presence of a trash item in a given room N */
+trash_at_room(N) [source(explorerAgent)]
  :  ~battery_charge(low) & .desire(go_to_room(M)) & N \== M
<-  .drop_desires(go_to_room(M)); /* Drop his current desire */
    !go_to_room(N);                 /* Commit a new intention */
    myLib.epistemic_action(N).     /* Perception action to locate trash items */

/* If the epistemic action succeeds to locate a trash item, it accordingly adds a new Belief. Such a belief is pivotal for agents intentions and determines the next course of actions (Goal Supporting Belief)*/
+trash(Tx,Ty)
  <-  !clean_trash(Tx,Ty). /*commit a new intention to clean the discovered trash*/

/* clean_trash Plan */
+!clean_trash(Tx,Ty)
  :  myLoc(Tx,Ty) & mySensor(S) & artifactID(envBoard,Env)
  <-  cartago.use(Env,cleanTrash(X,Y),S); /* Use a cartago artifact to perform the clean action */
    cartago.sense(Env, cleaned(X,Y), 100);
    -trash(Tx,Ty).                  /* Belief update */
+!clean_trash(X,Y)
  <-  !go_to_XY(loc(X,Y));
      !clean_trash(X,Y).
Cleaner Agents in Jason (II)

/* go_to_room Plan. Find a path to reach the desired room from the actual location and goes through it */
+!go_to_room(N)
  : not myRoom(N) & myLocation(X,Y)
  <- myLib.get_coords(X,Y, Coords); /* Coords is a list, whose terms are locations indicating the course to follow to reach the desired target */
    !go_to_XY(Coords); /* The terms of Coords are represented in the form loc(X,Y)*/
    -+myRoom(N). /* Belief Update */

/* go_to_XY Plan. Moves executing internal actions operating on the cartago envBoard */
+!go_to_XY([]).
+!go_to_XY([loc(X,Y)|T])
  : mySensor(S) & artifactID(envBoard,Env)
  <- cartago.use(Env, move(X,Y), S); /* Use a cartago artifact to move to a location */
        cartago.sense(Env, moved(X,Y), 100);
    -+myLocation(X,Y); /* Belief Update */
    !go_to_XY(T).
Customizing Agent Architecture

Users can specify a different overall architecture for an agent. Rather than producing a new reasoning cycle, this is used to customise the way in which the agent:

- does perception of the environment
- receives communication massages
- does belief revision
- acts in the environment
Redefining Agent Architecture

In the configuration file:

```java
agents: myAgent agentArchClass MyArch;
```

Example (customised architecture class)

```java
import jason.architecture.*;
public class MyAgArch extends CentralisedAgArch {
    public void perceive() {
        System.out.println("Getting percepts!");
        super.perceive();
    }
}
```
package alice.c4jason;

public class CAgentArch extends AgArch {
    ...
    public List<Literal> perceive()
        synchronized(cartagoEventQueue) {
            for (Literal l: cartagoEventQueue) {
                Trigger te = new Trigger(TEOperator.add, TETYPE.belief, l);
                getTS().getC().addEvent(new Event(te, Intention.EmptyInt));
                logger.fine("CAgent:buf - added event "+1);
            }
            cartagoEventQueue.clear();
        }

    synchronized(obsArtifactProperties) {
        if (obsArtifactPropertiesChanged) {
            ArrayList<Literal> newlist =
                new ArrayList<Literal>(obsArtifactProperties);
            obsArtifactPropertiesChanged = false;
            return newlist;
        } else {
            return null;
        }
    }

    ...
}
Customizing Agent Class

- Users can define a specific Agent Class, too.
- This is used to customise the selection functions of the AgentSpeak interpreter and other agent-specific functions:
  - Belief Revision
  - Event selection
  - Message and action-feedback (from environment) processing priorities
  - Applicable plans selection
  - Intention selection
  - Action selection
Redefining Agent Class

In the configuration file:

agents: myAgent agentClass MyAgentClass;

Example (customised agent class)

```java
import java.util.*;
import jason.asSemantics.*;
public class MyAgClass extends Agent {

    public Event selectEvent(List evList) {
        System.out.println("Selecting an event from "+ evList);
        return (Event)evList.remove(0);
    }
}
```
Conclusions

AgentSpeak
- Goal Oriented notion of Agency
- Mentalistic Notions as building blocks
- Agent programming
- Logic + BDI
- Operational Semantics

Jason
- AgentSpeak interpreter
- implements the operational semantics
- Support for Agent Communication Language
- Integration with CArtAgO to work in artifact based Environments
- Highly customisable, open source


From agents to artifacts back and forth: Purposive and doxastic use of artifacts in MAS.

Cognitive use of artifacts: Exploiting relevant information residing in MAS environments.
1st International Workshop (KRAMAS 2008), Sydney, Australia, 17 September 2008, Revised Selected Papers.

AgentSpeak(L): BDI agents speak out in a logical computable language.


Programming Intentional Agents in AgentSpeak(L) and Jason

Multiagent Systems LS
Sistemi Multiagente LS

Andrea Omicini & Michele Piunti
{andrea.omicini, michele.piunti}@unibo.it

Ingegneria Due
ALMA MATER STUDIORUM—Università di Bologna a Cesena

Academic Year 2009/2010