Programming Intentional Agents in AgentSpeak(L) and Jason

Laboratory of Multiagent Systems LM
Laboratorio di Sistemi Multiagente LM

Elena Nardini
elena.nardini@unibo.it

Ingegneria Due
Alma Mater Studiorum—Università di Bologna a Cesena

Academic Year 2010/2011
1 BDI Architecture
   • A BDI Model
   • BDI Implementations

2 AgentSpeak(L)

3 Jason

4 Exercises
   • Exercise 1
   • Exercise 2
   • Exercise 3

5 Conclusion

6 Bibliography
Outline

1. BDI Architecture
   - A BDI Model
     - BDI Implementations
2. AgentSpeak(L)
3. Jason
4. Exercises
   - Exercise 1
   - Exercise 2
   - Exercise 3
5. Conclusion
6. Bibliography
Basic Architecture of a BDI Agent [Wooldridge, 2002]
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
Loop Explanation

1. The agent initializes the internal states
2. The agent enters the main loop
3. The option generator reads the event queue and returns a list of options
4. The deliberator selects a subset of options to be adopted and adds these to the intention structure
5. The intentions to be adopted are filtered from the selected ones
6. If there is an intention to perform an atomic action at this point in time the agent executes it
7. Any external events that have occurred during the interpreter cycle are then added to the event queue (the same for internal events)
8. The agent modifies the intention and the desire structures by dropping successful ones
9. Finally, impossible desires and intentions are dropped too
Outline

1. BDI Architecture
   - A BDI Model
   - BDI Implementations

2. AgentSpeak(L)

3. Jason

4. Exercises
   - Exercise 1
   - Exercise 2
   - Exercise 3

5. Conclusion

6. Bibliography
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 form the interaction with external entities (i.e. message arrival, signals, etc.)
6. A **plan library** (repertoire of actions) as a further (static) component.
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
AgentSpeak(L)

- 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.
Jason [Bordini et al., 2007]

- 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

Jason [Bordini et al., 2007]
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)
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 \( \bar{g}(t) \) are goals, then
\[ +b(t), -b(t), +!g(t), +\bar{g}(t), -!g(t), \text{ and } -\bar{g}(t) \]
are \textit{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
- 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)
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:

```
triggering_event : context <- 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).
Jason Architecture
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
Jason as an Agent Programming Language

- Jason include all the syntax and the semantics defined for AgentSpeak
- Boolean operators
  - ==, <, <=, >, >=, &, |, \==, not
- Arithmetic
  - +, -, /, *, **, mod, div
- Then Jason includes several extesions
- 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 \( s_i \in \{ \text{self}, \text{percept} \} \cup \text{AgId} \)
  - \( \text{myLocation}(6,5)[source(self)]. \)
  - \( \text{red}(\text{box1})[source(\text{percept})]. \)
  - \( \text{blue}(\text{box1})[source(\text{ag1})]. \)

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

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

```prolog
+!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);
...

+!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)[deg0fCert(D1)]
    & not (colour(Obj,_,)[deg0fCert(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:**

```prolog
+!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

```
@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*. 
- 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 laboratory, 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.
Installing Jason

- Download Jason 1.3.3 from http://sourceforge.net/projects/jason/files/
Outline

1. BDI Architecture
   - A BDI Model
   - BDI Implementations

2. AgentSpeak(L)

3. Jason

4. Exercises
   - Exercise 1
   - Exercise 2
   - Exercise 3

5. Conclusion

6. Bibliography
Exercises

Exercise 1

Jason for JEdit

Further Jason functionalities are available in the menu Plugin->Jason.
Jason Project I

New Project

New Jason Project

New project parameters

Project name: factorial

Infrastructure: Centralised

Root location: /Users/elena/Desktop

Directory: /Users/elena/Desktop/factorial

Ok Cancel
Exercise 1

Jason Agent I

Elena Nardini (Università di Bologna)

AgentSpeak(L) & Jason

A.Y. 2010/2011 38 / 59
// Agent fact in project factorial.mas2j

/* Initial beliefs and rules */

print_fact(5).
+!fact(N,1) : N == 0.

+!fact(N,F) : N > 0
  <- !fact(N-1,F1);
  F = F1 * N.

/* Plans */

+!print_fact(N)
  <- !fact(N,F);
  .print("Factorial of ", N, " is ", F).
Project Running

/* Plans */

+!print_fact(N)
  <- !fact(N,F);
  .print("Factorial of ", N, " is ", F).

[fact] Factorial of 5 is 120

Running project factorial
Outline

1. BDI Architecture
   - A BDI Model
   - BDI Implementations

2. AgentSpeak(L)

3. Jason

4. Exercises
   - Exercise 1
   - Exercise 2
   - Exercise 3

5. Conclusion

6. Bibliography
Domestic Robot I

Requirements

- A domestic robot has the goal of serving beer to its owner.
- Its mission is quite simple, it just receives some beer requests from owner, goes to the fridge, takes our a bottle of beer, and brings it back to the owner.
- The robot should be also concerned with the beer stock and some rules hard-wired into the robot by the Department of Health.
Domestic Robot II

Agents
- Robot
- Owner
- Supermarket

Perceptions
- at(robot, Place). Only two places are perceived, fridge and owner. Thus, depending on its location in the house, the robot will perceive either at(robot, fridge) or at(robot, owner).
- stock(beer, N). When the fridge is open, the robot will perceive how many beers are stored in the fridge.
- has(owner, beer). It is perceived by the robot and the owner when the owner has a (non-empty) bottle of beer.
Supermarket Agent

last_order_id(1). // initial belief

// plan to achieve the goal "order" for 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)).
Owner Agent I

!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).
Owner Agent II

// 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).
/* Initial beliefs */

// initially, I believe that there is some beer 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 has been 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.
Robot Agent III

@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)).

@m1
+!at(robot,P) : at(robot,P) <- true.

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

// when the supermarket makes a delivery, 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).
Domestic Robot Running

![Diagram of a domestic robot environment with a fridge and owner locations marked on a grid.]

Fridge (3)  
Owner (10)
Outline

1. BDI Architecture
   - A BDI Model
   - BDI Implementations

2. AgentSpeak(L)

3. Jason

4. Exercises
   - Exercise 1
   - Exercise 2
   - Exercise 3

5. Conclusion

6. Bibliography
Thermostat Agent

Requirements

- Check the environment temperature $T$.
- Until $T$ is not: $> 18$ and $< 22$:
  - Decrease $T$ of one unit if the temperature is 22
  - Increase $T$ of one unit if the temperature is 18

Constraint

- No environment
- No interaction with other agents
Questions

- Is Agent programming well supported in Jason?
- Centralised or distributed Agents?
- Agent Interactions?
- Environment?


Programming Intentional Agents in AgentSpeak(L) and Jason

Laboratory of Multiagent Systems LM
Laboratorio di Sistemi Multiagente LM

Elena Nardini
elena.nardini@unibo.it

Ingegneria Due
Alma Mater Studiorum—Università di Bologna a Cesena

Academic Year 2010/2011