BDI Agent Programming with AgentSpeak

Michael Wooldridge
(mjw@liv.ac.uk)
What is AgentSpeak?

- A simple but powerful programming language for building *rational agents*
- Based on the *belief-desire-intention* paradigm
- Intellectual heritage:
  - The Procedural Reasoning Systems (PRS)
    - developed at SRI in late 1980s
  - Logic Programming/Prolog
What is Jason?

- An implementation of AgentSpeak
- A development environment for AgentSpeak systems
- Implemented in Java, has lots of hooks to call Java code
- Comes with libraries and debugging tools
- Get “up and running” very quickly
The AgentSpeak/PRS Architecture

- beliefs
- intentions
- plans
- desires
- events
- actions

Java code calls
AgentSpeak Control Loop

- agent receives *events*, which are either
  - external (from the environment, from perceptual data)
  - internally generated
- tries to *handle* events by looking for *plans* that *match*
- the set of plans that match the event are *options/desires*
- chooses one plan from its desires to execute: becomes committed to it -- an *intention*
- as it executes a plan may generate new events that require handling
The AgentSpeak Architecture: Beliefs

- beliefs in AgentSpeak represent information the agent has about its environment
- they are represented symbolically
  - ground atoms of first-order logic
The AgentSpeak Architecture: Example Beliefs

open(valve32)

father(tom, michael)

father(lily, michael)

friend(michael, john)

at_location(michael, gunne)

on(blockA, blockB)
The AgentSpeak Architecture: Plans

- coded by developer offline, in advance
- give the agent information about
  - how to respond to events
  - how to achieve goals
- plan structure:
  - event
  - context
  - body
triggerCondition :
  context <-
  body.
The AgentSpeak Architecture: Plan Structure

- **triggerCondition**
  - is an *event* that the plan can *handle*
- **context**
  - defines the conditions under which the plan can be used
- **body**
  - defines the actions to be carried out if the plan is chosen
The AgentSpeak Architecture: Events

- +! P
  - new goal acquired -- “achieve P”
- -! P
  - goal P dropped
- + B
  - new belief B
- - B
  - belief B dropped
Hello World

- Set up an empty directory called “hello_world” in your workspace
- Create a new project, called hello_world
  - to do this, use the “new project” button on JEdit
- Jason will create a template MAS folder
The Template MAS

/* Jason Project */

MAS hello_world {
    infrastructure: Centralised
    agents:
}

What does this say?

- It says that the system is called “hello_world”
- It says that currently, it contains no agents
- So let’s add some agents...
Add An Agent

• Use the button “add agent in project”
• Give it the name “hello”
• Again, Jason will produce a template with the “hello world” agent in
• if it doesn’t type this in.
The Hello World Agent

// Agent hello in project hello_world.mas2j

/* Initial beliefs and rules */

/* Initial goals */

!start.

/* Plans */

+!start : true <- .print("hello world.").
About the Hello World Agent

• The agent has a single *initial goal: !start*
  • this goal is there when the agent starts up
  • The exclamation mark says “this is a goal”
  • There is a single plan, which says “if you have acquired the goal “start”, then print “hello world”
  • Run the system by pressing the “play” button
Running and Debugging

- A console will open, which will show the output of all agents.
- It should show:
  - [hello] hello world.
- Congratulations!
- Press the “debug” button on the console to see inside the agent’s heads.
- Notice you have to explicitly stop the system from the jEdit console.
Plans

• A plan has the form
  • triggering_event : context <- body

• meaning
  • if you see this “triggering_event”
  • and believe the “context” is true
  • then you can execute “body”
A More Complex Example

• Create a new project “factorial1”, with a single agent “factorial1”
The Agent “factorial I”

fact(0, 1).

+fact(X, Y)
  :  X < 5
  <- +fact(X+1, (X+1)*Y).

+fact(X, Y)
  :  X == 5
  <- .print("fact 5 == ", Y).
• Initial belief says “the factorial of 0 is 1”
The First Rule

+fact(X,Y)
  :  X < 5
<-  +fact(X+1, (X+1)*Y).

- If you acquire the belief that the factorial of X is Y, and X is less than 5, then add the belief that the factorial of X+1 is (X+1)*Y
The Second Rule

+fact(X,Y)
  :  X == 5
<- .print("fact 5 == ", Y).

• If you acquire the belief that the factorial of X is Y, and X == 5, then print “fact ...”

• Notice the use of “==”.

• Don’t use “=” as it means something different

• Run the program and explore the agent’s mind
Inside the agent’s mind

- Here are all the beliefs the agent has accumulated.
- [source(self)] is an *annotation*, indicating where the belief came from...
- we will see how to use these shortly
A Small Modification

- Modify the agent so that intermediate results are printed as they are generated
Internal Actions

- `.print(...)` is an *internal action*
- other internal actions:
  - `.stopMAS()` -- stop system running
  - `.time(H,M,S)` -- put time into vars H,M,S
  - `.wait(X)` -- pause for X milliseconds
  - `.random(X)` -- put random value into X (0 <= X <= 1)
Further Modifications

- Modify your solution so that after the value is printed, the system pauses 3 seconds and then terminates.
- You should see the console displayed for 3 secs then disappear...
A Data Driven Solution

• Notice that the solution we have developed is data driven/ event driven

• It is the arrival of a partial solution that causes another partial solution to be generated...

• We can also look at a goal driven solution
factorial2

• Create a new project, “factorial2”, and within it a single agent “factorial2”

!print_fact(5).

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

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

+!fact(N,F) : N > 0
  <- !fact(N-1,F1);
  F = F1 * N.
Here the agent starts with a single goal, which is to print the factorial of 5

The first rule says, if you have this as a goal, then
- first compute the factorial of N
- then print it

The second and third rules say how to compute the factorial of N
Communication

- One agent is boring! Let's add more!
- We'll have an agent that knows how to compute factorial, and another that doesn't
- The expert will receive queries from the idiot and will respond to them
The `.send(...)` Action

- The basic mechanism for communication is the `.send(...)` action:
  
  `.send(rcvr, type, content)`

- Causes a message to be sent to agent called “rcvr”, with message type “type”, and content “content”
Example

- `.send(mjw, tell, fact(3, 6))`
  - this will cause the agent mjw to add the belief fact(3, 6)
- `.send(mjw, achieve, go(10, 10))`
  - causes +!go(10, 10) to be added as an event for mjw
- Actually its more complicated than that: the recipient annotates with the source
The Client-Server

• Create a new project, “factorial3”, with 2 agents: idiot and expert
The Idiot Agent

// Agent idiot in project factorial3.mas2j

/* Initial goals */

!start.

/* Plans */

/* Plans */

+!start : true
   <- .print("starting..");
   !query_factorial(2);
   !query_factorial(4);
   !query_factorial(6);
   !query_factorial(10).

+!query_factorial(X) : true <-
   .send(expert,tell, giveme(X)).

+fact(X,Y) : true <-
   .print("factorial ", X, " is ", Y, " thank you expert").
Another Modification

- Modify the idiot agent so that it:
  - starts by asking for the factorial of 0
  - as soon as it gets a reply for the factorial of $X$, waits 2 seconds and then asks for the factorial of $X+1$.
  
- You will have to kill this when it runs and runs...