5. AgentSpeak – Mars cleaning

In the last few lessons, we studied the communication protocols, ontologies and Jade. Today, we will move, from multi-agent programming point of view, to a higher level and will talk about a special multi-agent programming language – AgentSpeak. This language is designed mostly for implementation of BDI agents. Moreover, it also provides simple definitions of environment, where the agents can move and therefore it is possible to create rather simply even more complicated simulations.

AgentSpeak basics

AgentSpeak is an interpreted language with a Prolog-like syntax and functionality. Its interpreter is called Jason and can be downloaded from http://jason.sourceforge.net/wp/. The package also contains several examples.

Exercise 17 Download Jason from the link above and install it (unzip the archive).

In order to work with Jason you need an IDE. The distribution of Jason contains a version of jEdit, which supports the AgentSpeak and Jason projects. You can also install an Eclipse plugin, which adds Jason support. The installation instructions for the plugin are available at http://jason.sourceforge.net/mini-tutorial/eclipse-plugin/.

Exercise 18 Install the Eclipse plugin or start jEdit. Open one of the examples, start it and see what happens.

The Jason project consists of several things. The main file (.mas2j) describes the whole multi-agent system. Its syntax is rather simple, have a look at today source codes, you should not have any problems understanding it.

Exercise 19 Have a look at the mas2j file from today source. What do you see?

The mas2j file contains, apart from the list of agents, the name (and parameters) of the class which implements the environment, where the agents live.

Mars cleaning robots

Today, we will experiment with one of the examples provided with Jason. You can find it in the `examples/cleaning-robots` directory and a slightly updated version in today source codes.

The example contains two robots. One of them moves through the environment and collects garbage, the other one sits in the middle and destroys all the garbage the former robots takes to it (more precisely, all garbage at its location).

The environment is represented by a 7x7 grid and provides a few basic actions to the agents.

- `next(slot)` – moves the agent to the next slot
- `moveTowards(x,y)` – moves the agent one slot closer to the given coordinates
- `pick(garb)` – picks the garbage (can fail)
- `drop(garb)` – drops the garbage
- `burn(garb)` – burn the garbage at the `r2` location (that's the agent which burns the garbage)

The environment also provides the following information

- `pos(R,X,Y)` – robot R is at position (X,Y)
- `garbage(R)` – there is garbage at the position of robot R

Creating the environment

The environment for the agent is created as an implementation of the Environment class. It is important to implement the methods `init(String [])` and `executeAction(String ag, Structure action)`. The environment typically uses a model, in this case, a `GridWorldModel` which can be used for environments implemented as grids.

Exercise 20 Have a look at the implementation of the environment in today source codes.

Jason can also work with other frameworks for the implementation of more complicated environments, such as Cartago (http://cartago.sourceforge.net/). However, we will not discuss these framework any deeper.

Creating an AgentSpeak agent

Let's now have a look at AgentSpeak. Each AgentSpeak program consists of several parts – rules, initial beliefs, and plans and goals.

Rules and beliefs

Rules in AgentSpeak express relationship between beliefs, or any other general information about the world. For example, if we want to write a rule, which holds, if the robot `r1` is at the same location as robot `P` (and the locations of robots are represented as `pos(R, X, Y)` in beliefs) you can define a predicate `at(L)`.

```
    at(P) :- pos(P,X,Y) & pos(r1,X,Y).
```

Note that the syntax of the rules is similar to the syntax of Prolog. The only difference is that instead of comma the atoms are delimited by `&`. The AgentSpeak interpreter even makes unifications when applying a rule and the reasoning is also similar to Prolog.

Both rules and atoms the agents is provided in the beginning form the set of initial beliefs, which can later change during the life of the agent. The agent can act and reason based on these rules and beliefs.

Beliefs are automatically updated which the agents learns a new information from the environment or from another agent. Each belief can also have an annotation (written in square brackets) which describes the source of the belief, e.g. `likes(peter, music)[source(martin)]`, tell that the agent beliefs peter likes music, because the agent martin told him so. Annotations can be unified in the same way as beliefs.

Plans

The behavior of an agent is given mostly by its plans. The description of plan in Jason is again very simple. Each plan consists of three parts – trigger, context and the body.

The trigger tells when the plan should start. Syntactically, it is similar to the head of a predicate in Prolog with a few symbols added in front of the head. These symbols express whether the plan should execute after the addition (+) or removal (-) of a belief (nothing) or goal (!, or ? for query goals). If you have a look at the sample agent, you can find a plan which is executed if the `garbage(r1)` belief is added. Its trigger is `+garbage(r1)`. You can also find a trigger for the addition of the initial plan `check(slots)`, which is `+!check(slots)`. Again, unification is used with triggers in the same way as with the rules. The unified variables are bound in both the context and the body of the plan.

Next part of plan is a context (divided by : from the trigger). It is in fact a condition which must hold before the plan can be executed. The syntax is similar to the syntax of the body of a rule. Again, in the sample agent, you can find the rule

```
    +!ensure_pick(S) : garbage(r1)…
```

The part after the colon is the context.

There are also contexts, which contain the keyword `not`. It means, that the agent does not believe the condition is true. There is also the ~ operator, which means that the agent believes the condition is not true.

Exercise 21 Remember the difference between the closed world and the open world. What is the different between the two operators (`not` and ~)?

The last part of each plan is its body. It is delimited by <− from the context and contains a list of actions the agent should perform. There are several kinds of actions. The actions provided by the environment are written without any qualifier (e.g. `pick(garb)`), on the other hand, internal actions of the agents start with a dot (`.print("Hello world")`).

There are special types of actions to add a new plan (they start with ! or ? for query plans), add (+) or remove (-) a belief. We can also use (-+) as a simplification for the update of a belief. If you want to ask the belief base, you can use the ? operator (it is in fact addition of a query goal).

The query goal is a special type of plan. It is handy, if we need to find out something, which is not in the belief base (e.g. an agent sitting in the living room wants to know, whether there is beer in the fridge, it needs to move to the fridge first).

Exercise 22 Examine the sample agent and try to understand what it does.

Exercise 23 Change the environment to include two agents burning the garbage.

Exercise 24 Change the cleaning agent to take the garbage to the closes burner.

Exercise 25 If you have more time, add more cleaning agents.

What have we learned today

1. Description of environment in Jason
2. AgentSpeak syntax
3. Goals and plans of agents, belief base