Robotics and Autonomous Systems

Lecture 20: More Complex Programs in AgentSpeak and Jason

Richard Williams

Department of Computer Science
University of Liverpool

Today

- In this lecture we will look in more detail at the tools that you will use for the second assignment:
  - AgentSpeak
  - Jason
- AgentSpeak is a programming language.
- Jason is an environment for building agents.
- They can be combined with Java/LeJOS for building robot controllers.

HelloWorld in Jason

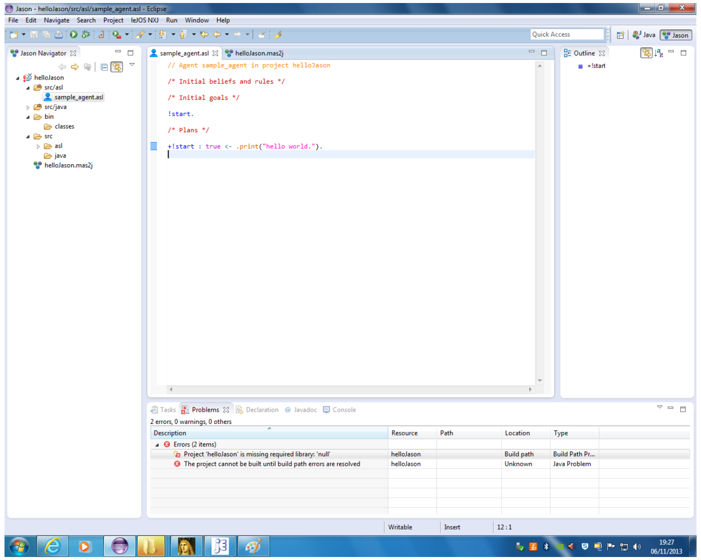
- Create a Jason project "helloworld" in Eclipse, and you get:

```
MAS helloworld{

   infrastructure: Centralised

   agents:
      agent1 sample_agent;

   aslSourcePath:
      "src/asl";
}
```


HelloWorld in Jason

- `infrastructure`: how the agent system is organised.
- `agents`: the list of agents that make up the system.
  Here there is just one.
- `aslSourcePath`: path from the MAS file to the agent descriptions.

HelloWorld in Jason



HelloWorld in Jason

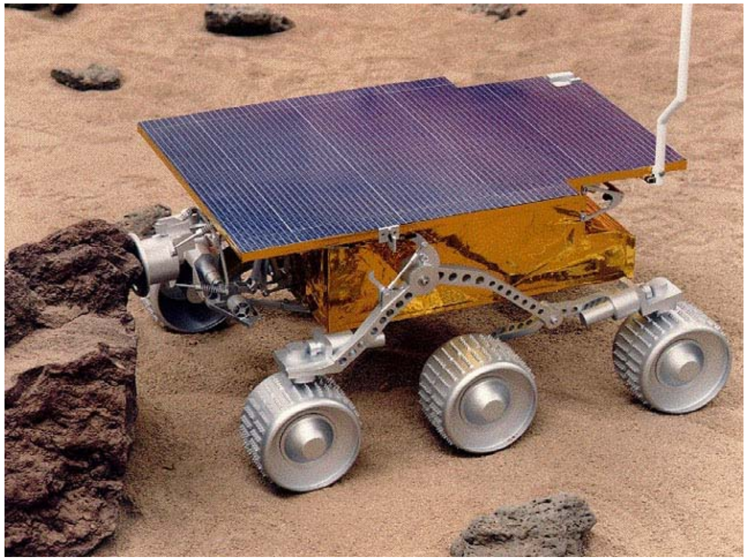
- The agent looks like this:

```
/* Initial beliefs and rules */

/* Initial goals */

!start.

/* Plans */

+!start : true <- .print("hello world.").
```

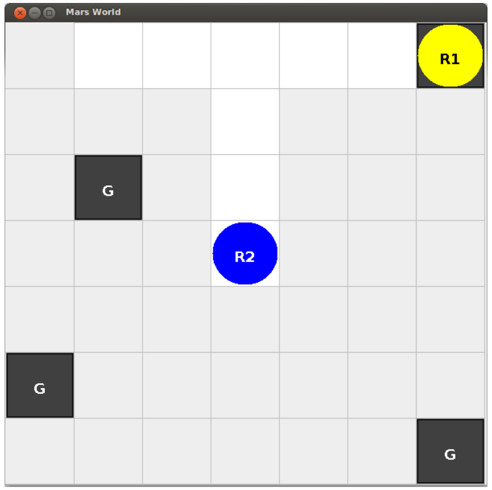

HelloWorld in Jason

- No initial beliefs or rules
- Only goal is the achievement goal `start`.
- The context/precondition for `start` is `true`.
- The plan for `start` is to print “hello world.”

Mars Rover example



Environment



- This is the `cleaning-robots` example from the Jason distribution.

Garbage collection

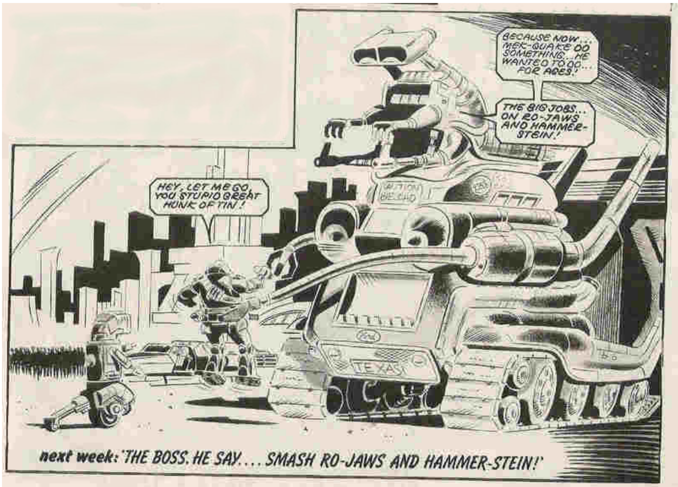
- `r1` collects the garbage.

Garbage collection

- DustCart in Peccioli

Peccioli

Garbage disposal



- `r2` disposes of the garbage.

How do we do this?

- Set up MAS.
- Set up environment.
- Set up robots.

MAS

- Here is a suitable MAS description.

```
MAS mars {

    infrastructure: Centralised

    environment: MarsEnv

    agents: r1; r2;
}
```

- `mars.mas2j`

MAS

- Environment is defined by the `MarsEnv.java` file.
- Defines several primatives that the agents can execute which complete actions in the environment or sense the environment.

MAS

- Environment is defined by the `MarsEnv.java` file.
- Defines several primatives that the agents can execute which complete actions in the environment or sense the environment.
- In your assignment, the robot will provide this.

A program for r1

- We will build up the program bit by bit.
- First, a program to move around the world.
- The environment calls each square a “slot”, and provides the primitive:
  `next(slot)`
  to move from one to another.
- The environment also provides the position of the robots through the predicate:
  `pos(robot, xloc, yloc)`
- A simple program to move through the space is the following.

A program for r1

- The initial goal for all these programs is:

```
/* Initial goal */

!check(slots).
```

- This has no inherent meaning, just a high level goal that coincides with the head of a plan.

Version 1

```
/* Plans */

// Step through the gridworld and then stop
//
// To achieve the goal !check(slots): if the robot
// isn't at the end of the world, move to the next
// slot, then reset the goal !check(slots)
+!check(slots) : not pos(r1,6,6)
   <- next(slot);
      !check(slots).
// Achieve the goal !check(slots) without doing
// anything.
+!check(slots).
```

- This is the program `r1_v1.asl` on the course website.

Version 2

- We have a slightly different version of `+!check(slots)`

```
// Step through to the first piece of garbage
//
// In this version, we keep moving so long as
// we don't sense garbage.
+!check(slots) : not garbage(r1)
   <- next(slot);
      !check(slots).
+!check(slots).
```

- So the stop condition is finding garbage rather than getting to the end of the world.

Version 2

- This version also says what to do if we sense garbage. If there is a belief event `garbage(r1)`.

```
+garbage(r1) : true
   <- .print("Garbage!").
```

- This is the program `r1_v2.asl` on the course website.

Version 3

- We want version 3 to pick up the garbage when it finds it.
- `!check(slots)` is the same:

```
+!check(slots) : not garbage(r1)
   <- next(slot);
      !check(slots).
+!check(slots).
```

- But the garbage handling part needs to be altered:

```
+garbage(r1) : true
   <- .print("Garbage!");
      !ensure_pick(garb).
```


Version 3

- Picking up garbage is not deterministic, so we need a recursive plan to make sure it happens.
  `pick(garb)` is another primitive.
- We keep trying to pick it up until we succeed.

```
+!ensure_pick(G) : garbage(r1)
   <- pick(garb);
      !ensure_pick(G);
  !check(slots).
+!ensure_pick(_).
```

- Then we continue moving.
- The last clause gives a way of achieving the goal when there is no garbage (the robot does nothing).

Version 3

- With the previous version of `!check(slots)` this will collect garbage, but get stuck at the end of the grid, trying to move forward.
- To prevent this, we add the following *before* the first clause of `!check(slots)`:

```
+!check(slots) : not garbage(r1) & pos(r1, 6, 6).
```

- This prevents the recursive call if the robot is at the end of the grid.

Version 3

- The full code for `!check(slots)` is then:

```
+!check(slots) : not garbage(r1) & pos(r1, 6, 6).
+!check(slots) : not garbage(r1)
   <- next(slot);
      !check(slots).
+!check(slots).
```

- All of this is the program `r1_v3.asl` on the course website.

Version 4

- To make the robot go to `r2` to dispose of the garbage, we need to first modify what we do when we find garbage.

```
+garbage(r1) : true
   <- .print("Garbage!");
      !take(garb,r2).
```

- When we find garbage we take it to `r2`.

Version 4

- To take the garbage to `r2` we make the robot at the location of `r2` and then drop the garbage.
  `drop(garb)` is another primitive.

```
+!take(G,L) : true
   <- !ensure_pick(G);
      !at(L);
      drop(G).
```

- We need two things to make this work.

Version 4

- First we need how to compute the location of the robot from the `pos` primitive.

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

- This is added to the (currently empty) beliefs of the robot.

Version 4

- Then we say how we achieve the goal of being at a location.

```
+!at(L) : at(L).
+!at(L) <- ?pos(L,X,Y);
           move_towards(X,Y);
           !at(L).
```

- We do this by repeatedly moving one step towards the right location.
- The step is achieved using the primitive `move_towards(X,Y)`
- As before, we do the repetition by recursion.
- This is the code in `r1_v4.asl` on the course website.

Onwards

- At this point it is only a short step to the version that you can download with the Jason distribution.
- Right now `r1` stops when it gets to `r2` and drops the garbage.
- Adding another `!check(slots)` will kick it back into motion.

Onwards

- Unfortunately, this will start it off again from the location of `r2`.
- That misses some garbage.
- It also covers some parts of the grid more than once.
- To get around this we need to:
  - Remember where we picked up the garbage.
  - Go back there form `r2`
- The full version of the rover (which is on the course website) gives an elegant solution.

A challenge

- A good exercise is to go look at the full version, run it, and see if you understand why it behaves as it does.
- When the first assignment is complete, I will post a lab that has you modify the mars robot example as a way of getting to grips with AgentSpeak and Jason.

Summary

- This lecture looked at writing programs in AgentSpeak/Jason.
- We briefly recapped some of the basic material from last time.
- We looked again at “hello world!”.
- Then we launched into a larger Mars Rover example.
- We looked at several steps on the way to building a full implementation of this example.