SYSC 5103 – Software Agents

RoboCup Project

Motasem Bakieh and Itaf Omar Joudeh, SYSC 5103 Students

***Abstract*—In this project description paper, we will demonstrate how soccer-playing agents were implemented using an existing BDI agent framework to compete in an in-class RoboCup competition. AgentSpeak(L) is an agent-oriented programming language that is based on the BDI architecture [1]. Jason is the most regularly maintained BDI framework, which interprets AgentSpeak(L) [1]. Jason was used to develop and customize the behaviours as well as the architecture of an AgentSpeak(L) soccer-playing agent. Perception functions were extended using Java to help the agent perceive the current state of the environment, and act accordingly.**

Agent, Agent Architecture, AgentSpeak, BDI, Eclipse, Jason, Java, RoboCup, Soccer, Player

I. Introduction

RoboCup is an international scientific competition where teams of different programs compete against each other. Using the RoboCup simulation league, percepts and actions, which depend on the ball and opponent's goal locations with respect to the soccer-playing agent, were generated. Jason is a well-known Java-based interpreter for an extended version of AgentSpeak(L) [1]. The customization of Jason's agent architecture class has enabled the bridge between Krislet's Java classes and the AgentSpeak(L) soccer-playing agent.

II. Beliefs-Desires-Intentions (BDI)

Building agent applications which include beliefs, desires, and intentions can play a motivational role in reasoning processes. In such applications, goals are achieved using the rational acts of the agents with knowledge of the environment. To implement a RoboCup soccer-playing agent using an existing BDI agent framework, a BDI representation of the problem was established. Table I below summarizes the applicable beliefs, desires, and intentions.

TABLE I
Summary of Beliefs, Desires, and Intentions

Beliefs	ball location: distance, direction
	opponent's goal location: direction
Desire	scoring a goal
Intentions	locating the ball and/or goal
	orienting to the ball
	approaching the ball
	kicking the ball

The ball location in terms of its distance and direction as well as the opponent's goal location in terms of its direction are depicted as percepts of the environment. Such percepts (beliefs) are then inputted

into the agent's selection functions as the informative component of the system. The agent's desire of scoring a goal is inputted into the agent's selection functions as the motivational state of the system. As a result, the agent outputs its intentions as a course of actions to fulfill its desire. The soccer-playing agent decides whether to turn, dash, or kick. Figure 1 below illustrates the interactions between the soccer-playing agent and the environment.

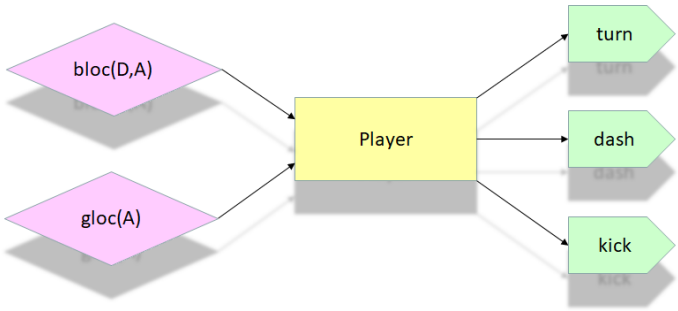

Figure 1: Player-Environment Interactions

III. CODES

Eclipse was used as the integrated development environment (IDE) to view and edit the code. Thus, Jason was used as an Eclipse plug-in. The next two sections include discussions about the developed AgentSpeak(L) agent, and Java classes.

A. AgenSpeak(L) Soccer-playing Agent

This section explains the program that was written in AgentSpeak(L). The scenario involves ten soccer-playing agents; five per team. Agent `player.asl` attempts to score a goal by following a set of intentions. Since the agent's actions depend on the location of the ball and/or goal at any given time, not on external factors, the environment is completely deterministic. The AgentSpeak(L) code for `player` is given below; we have annotated each plan with a label, so that we can refer to them in the text below.

```
!start.
+!start : true <- move.                                    (p1)

+bloc(D,A) : D == -1 & A == -1                             (p2)
  <- !locating(ball).

+!locating(ball) : bloc(D,A) & D == -1 & A
== -1                                                      (p3)
  <- turn;
     !locating(ball).
+!locating(ball) : true                                    (p4)
  <- true.

+bloc(D,A) : D > 1 & not(A == 0)                           (p5)
  <- !orienting(ball).

+!orienting(ball) : bloc(D,A) & D > 1 &
not(A == 0)                                                (p6)
  <- turn;
     !orienting(ball).

+!orienting(ball) : true                                   (p7)
  <- true.

+bloc(D,A) : D > 1 & A == 0                                (p8)
  <- !approaching(ball).

+!approaching(ball) : bloc(D,A) & D > 1 & A
== 0                                                       (p9)
  <- dash;
     !approaching(ball).

+!approaching(ball) : true                                 (p10)
  <- true.
+gloc(A) : A == -1                                         (p11)
  <- !locating(goal).

+!locating(goal) : gloc(A) & A == -1                       (p12)
  <- turn;
     !locating(goal).

+!locating(goal) : true                                    (p13)
  <- true.

+gloc(A) : not(A == -1)                                    (p14)
  <- !kicking(ball).

+!kicking(ball) : gloc(A) & not(A == -1)
  <- kick;                                                 (p15)
     !kicking(ball).

+!kicking(ball) : true                                     (p16)
  <- true.
```

The initial goal of a `player` is to 'move' somewhere on the field before the game is kicked

off. This is done via plan `p1`. All of the other plans are explained below.

Plans `p2` through `p4` together ensure that a `player` will keep 'turning' in order to locate the ball, until it can perceive the ball's distance and direction. Plans `p5` through `p7` together ensure that a `player` will keep 'turning' until it is oriented towards the ball (i.e. `direction = 0`). Plans `p8` through `p10` together ensure that a `player` will keep 'dashing' until it reaches the ball. Plans `p11` through `p13` together ensure that a `player` will keep 'turning' in order to locate the opponent's goal, until it can perceive the goal's direction. Plans `p14` through `p16` together ensure that a `player` will keep trying to 'kick' the ball until it succeeds. The only problem encountered while a `player` is turning is that it often turns fast and/or more than needed that it misses the ball.

B. RoboCup

Krislet was used as the starting point. The default Krislet behaviour was then modified by changing the `Brain` class. The `Brain` class now extends Jason's `AgArch` class and implements two other classes: `SensorInput` and `Runnable`. Furthermore, the `hear(int time, String message)` method was updated to receive information regarding the current play mode from the referee.

The `perceive()` and `act(ActionExec action)` methods of the `AgArch` class were overridden to send percepts to the agent, and to receive and execute the agent's actions, respectively.

The code is available online, on GitHub, at https://github.com/Itaf/SYSC5103. More specifically, the code can be found in the "RoboCupDemo" folder. To run the code:

1) Download as a zip, then extract the files
2) Start the server
   i. Go into "rcssserver-14.0.3-win"
   ii. Double click on "rcssserver.exe"
3) Start the monitor and connect it to the server
   i. Go into "rcssmonitor-14.1.0-win"
   ii. Double click on "rcssmonitor.exe"
4) Start the clients and connect them to the server
   i. Go into "Krislet"
   ii. Double click on "TeamStart.bat"
5) Use the monitor to begin the game
   i. Press on "Referee" from the menu bar
   ii. Select "KickOff"

The "TeamStart.bat" file runs five instances of the soccer-playing agent.

IV. Conclusion

A soccer-playing agent was written in AgentSpeak(L), which is based on the BDI architecture. A few functions were implemented to generate the agent's perception of the environment, and to perform the agent's actions on the environment. Software simulations of RoboCup allow for further research to analyze and enhance agent behaviours.

V. References

[1] Jason.sourceforge.net. (2018). Jason | a Java-based interpreter for an extended version of AgentSpeak. [online] Available at: http://jason.sourceforge.net/.