RoboCup Team Using the Jason BDI Framework

Tarek Dawoudiah, Zain-Ur-Rehman, Abubakar Irfan

Faculty of Engineering, Department of Electrical and Computer Engineering, University of Ottawa 75 Laurier Ave E, Ottawa, ON, Canada

ABSTRACT

This paper will aim to show the implementation of a RoboCup team using the Jason BDI (Beliefs-Desires-Intentions) Framework. Jason is an open-source interpreter for an extended version of AgentSpeak, which is a logic-based agent-oriented programming language written in Java [1]. The implemented agent will make decisions based on the RoboCup environment and the Jason BDI engine, which will show how the agents could perform with practical reasoning. Using Jason, we will demonstrate how the RoboCup player agent performs reasoning and scores a goal.

Index Words: RoboCup, Jason, BDI, Agent, Agent Speak, Eclipse

1. INTRODUCTION

RoboCup is an international scientific initiative established in 1997 with the goal to advance the state of the art of intelligent robots [2]. RoboCup has now become an annual international robotics competition aiming to promote robotics and artificial intelligence research [3].

This project will aim to show the implementation of a RoboCup team in a 2D playing environment, where each team will have five autonomous soccer playing agents. The autonomous soccer playing agents will connect to a server and the networking with the server can be done using a programming language. In this project the RoboCup team environment will use Java along with the Jason BDI software model. The Jason BDI is a software model used for programming intelligent software agents. It uses the agent's concepts of beliefs, desires, and intentions to solve a problem using agent programming [3]. The agent programming language we will use for this project is the AgentSpeak Language, which is an agent-oriented programming language that is based on the Jason BDI software architecture. The Java functions will be used to help the Jason BDI agent identify the current state of the environment and act according to the information.

The upcoming sections of the report will discuss the software design, architecture and implementation and provide test result cases. Lastly the report will also provide a set of clear instructions on how to execute the code.

2. DESIGN AND ARCHITECTURE

Figure 1 shows the architecture of the program. The green lines/boxes show the new and modified Java modules/classes added and integrated with existing RoboCup files to make the Jason BDI agent.

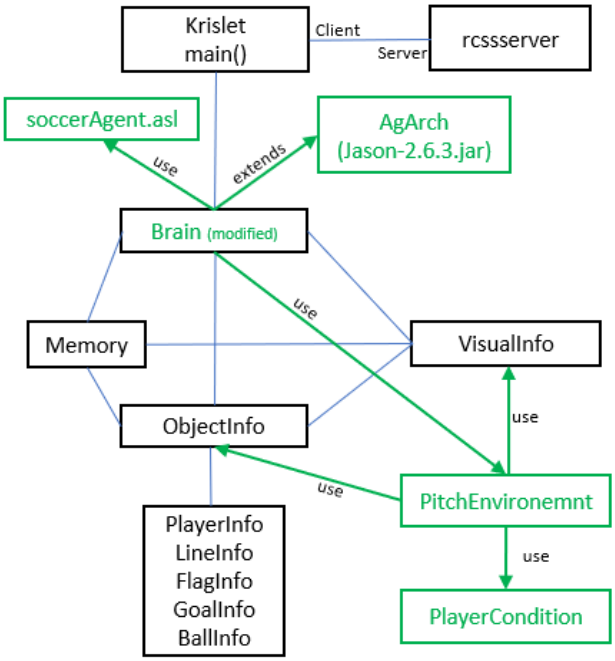

Figure 1

For simplicity, we assume that the reader is fully aware of the original RoboCup structure, hence we only mention the newly implemented functionality and how it's integrated with the existing RoboCup files. Our implementation is based on one of the Jason demos that demonstrate how a Java program is able to use the Jason AgentSpeak interpreter with a pre-made environment [4].

Important Jason architecture and semantics libraries are imported in Brain.java such as Agent, AgArch, ActionExec, and TransitionSystem for the BDI agent to work. RoboCup's Brain class is modified to use the functionalities and extends the Jason-2.6.3 AgArch class. Jason agent is initialized and soccerAgent.asl file is loaded. Using the PitchEnvironment class, we then initialize the local environment.

In the while loop of the Brain.java file, calls are made to the Jason engine to perform the reasoning cycle. The `reasoningCycle` function senses, deliberates, and acts based on the asl file. In order to sense, deliberate and act in the RoboCup environment, we had to override `perceive` and `act` functions.

The `perceive` function uses the newly implemented classes PitchEnvironment and playerCondition to perceive the environment conditions and return them in the appropriate fashion required by the Jason interpreter. The `act` function performs the actions on Krislet and lets player to kick, dash and turn based on the plans specified in soccerAgent.asl file and percepts received from the `perceive` function.

3. AGENT BEHAVIOUR

In this section, we will discuss our agent AI which is described in our asl file. An asl file contains the list of all the plans required for the agent to reach its goal. When these plans are triggered due to satisfied events (conditions in our environment), our agent performs actions that bring it closer to its goal.

The agent's main goal is for it to kick the ball at its opponent's goal, ultimately to score a goal. We defined two main plans that allow us to reach our goal: see and move. Plan see's main purpose is to make the agent see the ball when it's in a state where it cannot. Plan move's main purpose is to dash towards the ball, align with the opponent's goal, and shoot. The behavior of the agent is defined as:

1) Find the ball
2) Dash to ball
3) Get close enough to kick the ball
4) Aim at the opponent's goal
5) Kick the ball at the opponent's goal

Depending on our environment conditions (events), the BDI engine chooses the plan that is more specific and proceeds to execute it. Each plan eventually makes the agent perform a RoboCup action. The actions we perform are: kick, dash (value of dash is a variable depending on the environment), and turn (angle of turn depends on the plan executed).

The events we collect from the environment are described below:

- canSeeBall: The agent can see the ball.
- closeToBall: The agent is 1 meter or less away from the ball.
- aroundTheBall: The agent is between 1 and 3 meters away from the ball.
- farFromBall: The agent is more than 3 meters away from the ball.
- directlyFacingBall: The ball is within -10 and 10 degrees angle from the agent's view.
- canSeeOpponentGoal: The agent can see the opponent's goal.
- aimingAtOpponentGoal: The agent view is within the opponent's goal's posts.

Based on the observed events above and the plans selected, the following are the mapping between the asl actions and the Krislit actions:

- turn: Turn with 40 degree angle.
- turn_to_ball: Turn with angle of the ball.
- turn_to_goal: Turn with angle of the opponent's goal.
- dash_to_ball: Dash with speed of distance from ball multiplied by a factor of 5.
- kick: Kick the ball with strength 100 towards the opponent's goal.

Figure 2 shows all the plans, the events required to execute them, and the actions they perform when executed.

```
////////////////////////////// Inital goals //////////////////////////////
!see.
////////////////////////////// SEE plans //////////////////////////////
//Plan 1: Can't see the ball -> turn to find it
+!see
    :   not canSeeBall
    <-  turn;
        !see.
//Plan 2: Can see the ball -> dash towards it
+!see
    :   canSeeBall
    <-  dash_to_ball;
        -canSeeBall;
        !move.
//Plan 3: Back up scenario
-!see
    <-  turn;
        !see.
////////////////////////////// MOVE plans //////////////////////////////
//Plan 1: Can see the ball but it's still far -> dash towards it
+!move
    :   canSeeBall & farFromBall
    <-  dash_to_ball;
        -canSeeBall;
        -farFromBall;
        !move.
//Plan 2: Around the ball but not directly infront of it -> face the ball
+!move
    :   canSeeBall & aroundBall & not directlyFacingBall
    <-  turn_to_ball;
        -canSeeBall;
        -aroundBall;
        !move.
//Plan 3: Around the ball and directly facing it -> dash towards it
+!move
    :   canSeeBall & aroundBall & directlyFacingBall
    <-  dash_to_ball;
        -canSeeBall;
        -aroundBall;
        -directlyFacingBall;
        !move.
//Plan 4: Beside the ball and ready to kick but can't see the opponent's goal -> turn to find it
+!move
    :   closeToBall & not canSeeOpponentGoal
    <-  turn_to_goal;
        -canSeeBall;
        -closeToBall;
        !move.
//Plan 5: Beside the ball and ready to kick but can't see the opponent's goal -> turn to find it
+!move
    :   closeToBall & canSeeOpponentGoal & not aimingAtOpponentGoal
    <-  turn_to_goal;
        -canSeeBall;
        -closeToBall;
        -canSeeOpponentGoal;
        !move.
//Plan 6: Beside the ball and ready to kick -> kick at the goal
+!move
    :   closeToBall & canSeeOpponentGoal & aimingAtOpponentGoal
    <-  kick;
        -canSeeBall;
        -closeToBall;
        -canSeeOpponentGoal;
        -aimingAtOpponentGoal;
        !move.
//Plan 7: Can't see the ball anymore -> turn to find it
+!move
    :   not canSeeBall
    <-  turn;
        !see.
//Plan 8: Back up scenario
-!move
    <-  turn;
        !move.
```

Figure 2

When we first enter the environment, we initiate the see plan. This initiative plan is enough to get the agent workings towards its goal.

4. EXECUTION INSTRUCTIONS

The project source code has been submitted inside the zip folder along with the report and presentation. The code is extremely easy to run since the Jason class path has been defined inside the TeamStart.bat. Therefore, this allows the code to be run exactly like the original Krislet implementation. To run the code: Download the project zip file and extract the folder. Then go to the rcssserver folder and run the rcsssserver.exe executable. After that go to the rcssmonitor folder and run the rcssmonitor.exe executable. Then go to the Krislet folder and run the TeamStart.bat. After this wait for all the players to connect. Once the players are all connected, go to the referee tab on the monitor and click on KickOff.

5. TESTING AGENT BEHAVIOR

In order to test our agent, we created a new batch file that only initiates one RoboCup agent. The goal of the test was to see if the agent can locate the ball by turning, dash towards it and kick it towards the opponent's goal. After several modifications to the asl file, we were satisfied with the agent's behaviour after which we extended batch file to initiate a 5v5 game.

6. CONCLUSION

All in all, the project has been a success as we have successfully implemented a RoboCup team using the Jason BDI framework. As a future project, we want to implement new asl files to outline the behaviours of a defender and goalie. It will be interesting to see how a team consisting of attacker, defender and goalie will play against the team consisting of only attackers. This will be relatively easy to do by following the current architecture of the project design. Since the code is cleanly designed and well documented further additions are easy to make.

7. REFERENCES

[1] R. H. Bordini, Hübner Jomi Fred, and M. J. Wooldridge, *Programming multi-agent systems in AgentSpeak using jason*. Chichester, England: J. Wiley, 2007.

[2] “Robocup Federation Official Website,” RoboCup Federation official website, 17-Nov-2021. [Online]. Available: https://www.robocup.org/. [Accessed: 09-Dec-2021].

[3] “Belief-desire-intention software model,” Academic Dictionaries and Encyclopedias. [Online]. Available: https://en-academic.com/dic.nsf/enwiki/951131. [Accessed: 09-Dec-2021].

[4] Jason-Lang, “Jason/demos/using-only-jason-BDI-engine at master · Jason-Lang/jason,” GitHub. [Online]. Available: https://github.com/jason-lang/jason/tree/master/demos/using-only-jason-BDI-engine. [Accessed: 09-Dec-2021].