Robocup Project Report

Aron Arany-Takacs
Electrical Engineering and Computer Science
University of Ottawa
Ottawa, Canada
aaran043@uottawa.ca

Bardia Parmoun
Systems and Computer Engineering
Carleton University
Ottawa, Canada
bardiaparmoun@cmail.carleton.ca

Dylan Leveille
Systems and Computer Engineering
Carleton University
Ottawa, Canada
dylanleveille@cmail.carleton.ca

***Abstract*—JASON is a Java-based interpreter for an extended version of AgentSpeak. In this work, we make use of JASON to prescribe unique beliefs, desires, and intentions to different player types in a Robocup soccer team. Specifically, the player types that have been created using JASON are *Goalie*, *Defender*, and *Attacker*. The primary contribution of this work is the creation of distinct player types that possess a heightened sense of spatial awareness (e.g., using player and ball locations) to allow for better positioning during play. Such positioning allows the players to be better prepared against opponent strategies and increase their overall cooperation.**

I. Program Design

This section describes the design of our program. Our Robocup team is composed of five players: two *Attackers*, two *Defenders*, and one *Goalie*. Each player type is defined in their respective .asl file within the resources folder. This separation enables unique beliefs, desires and intentions for each player type.

Figure 1 provides an overview of the process followed by our program when initializing a new Robocup player. When starting a player (through RobocupAgent.java), depending on the player type provided as input for the player, the corresponding player type enum is sent to the Brain.java constructor. Brain.java uses this enum to identify the .asl file to be used for the player. Following this, it creates a JASON agent from the specification in this .asl file. Once finished, RobocupAgent.java will start the Brain.java thread. As Brain.java extends JASON's AgArch class, it will perceive and interact with the Robocup environment.

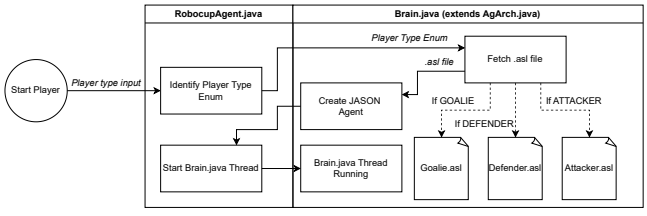

Fig. 1. An overview of the workflow for initializing Robocup player.

Figure 2 provides an overview of the process followed by our program during the execution of a Robocup player. When the Brain.java thread begins execution, depending on its player type, it will place the player in its respective playing position on the field. While the thread is running, a JASON reasoning cycle will be requested. The JASON transition system (specified in TransitionSystem.java) will request that the environment be perceived. To perceive the environment, the Perceive method was overridden in Brain.java. In this method, we perceive various environmental properties to create logical literals that serve as perceivable beliefs for the player. These literals are passed to the JASON transition system.

Using these perceivable beliefs, certain actions may be considered for certain steps in the intentions reasoned by JASON. If an action is required, the action is taken in the overridden act method of Brain.java. Specifically, in this method, based on a text-formatted action provided by JASON, an actual action corresponding to this string is taken by the player. When the reasoning cycle ends, if the player has performed an action during the reasoning cycle, the player will go to sleep for the duration of a Robocup clock cycle to avoid sending more than one player action to the Robocup server. When the sleep is finished, another JASON reasoning cycle will be requested. Similarly, if no action was taken during the reasoning cycle, a new JASON reasoning cycle will be requested.

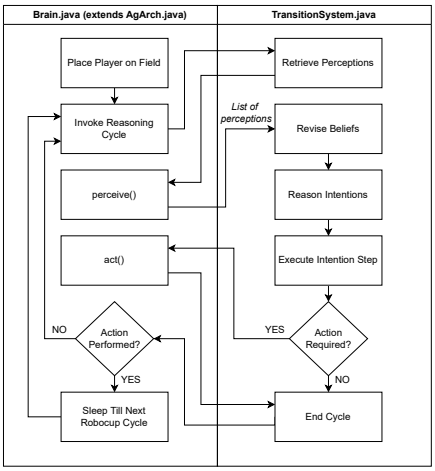

Fig. 2. An overview of the workflow during execution of a Robocup player.

II. Player Functionalities and Expected Behaviour

In this section, we describe each player type's functionality and expected behaviour.

A. Goalie

The goalie's `JASON` specification is found in the `goalie.asl` file. Fundamentally, the goalie should not move away from its net unless a ball is approaching and must be caught. As a result, in this file, notice that the goalie has an initial goal of `!wait`. With this goal, it simply attempts to find the ball and keep track of its direction. When the ball moves away from the center, and passes the center-top (`flag c t`) and center-bottom (`flag c b`) flags at certain angles, the goalie will move to different positions to better protect its goal. Specifically, there are five possible positions for the goalie: *center*, *right*, *veryRight*, *left*, and *veryLeft*. The goalie will start in the *center* position (explaining its initial belief, `in_centre_position`). When the ball has passed the center-top or centre-bottom flags at an angle of 5 degrees, it will move into the *right* or *left* position respectively (depending on the goalie's side of the field). Similarly, when the ball has passed these flags at an angle of 20 degrees, it will move into the *veryRight* or *veryLeft* position respectively (again, depending on the goalie's side of the field). Figure 3 labels and visualizes each possible goalie position.

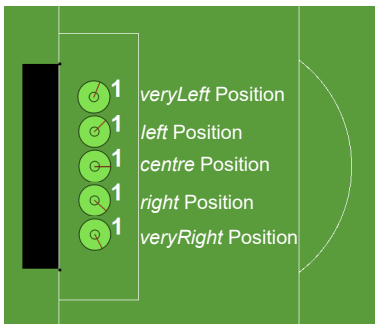

Fig. 3. A visualization of all possible goalie positions.

The goalie will wait in these positions until the ball is within a distance of 10 meters, and the ball is at a minimum angle of 7.5 degrees from the goalie. If the angle is too large, the goalie will attempt to turn to the ball to close the angle. When both of these conditions are met, the goalie will dash to the ball and attempt to catch it. If caught, the goalie will pass the ball to one of its teammates. Specifically, if there are no opponents between the goalie and the furthest defender on the team, the ball will be passed to that defender. Similarly, if there are no opponents between the goalie and the nearest attacker on the team, the ball will be passed to that attacker. If the ball cannot be passed to either of these players, it will be kicked to the right at a 45-degree angle from the center, near the center-top flag (`flag c t`). When the ball is no longer within 10 meters of the goalie, the goalie will reset to its *centre* position.

B. Defender

The defender's `JASON` specification is found in the `defender.asl` file. The defender will only have 3 main goals throughout the game. Its first goal, `!wait`, has to do with its main loop which is very similar to Krislet's behaviour. In other words, the defender first the player attempts to locate the ball through 40-degree turns. When found, it will continuously align with the ball. When the ball reaches a close distance (1 m) to the defender, the defender checks if it can view the opposing goal. If the goal is visible, it attempts a kick towards that goal. If not, it will look for an attacker on its team, and attempt a pass to that attacker. If the latter cannot be found, it will attempt to look for attackers on its team through 40-degree turns.

A key component of the defender is its ability to ensure that it remains in its operating zone. As a result, its second `JASON` goal is to return to its home zone once it exits it. To achieve this, a defender will repeatedly scan flags around the field and calculate their distance from each. On each side of the field, the defenders use that field's goal flag along with its corner flags. If the defender's distance to any of its home flags is below a threshold (18 m or 35% of the field length), it registers `in_home_zone` belief. Similarly, if the defender's distance from any of its opponent flags is below a threshold, it registers a `in_home_zone` belief. When the defender is not close to any of the flags, it will not know anything about its home zone.

Figure 4 illustrates these zone calculations for both the attacker and the defender.

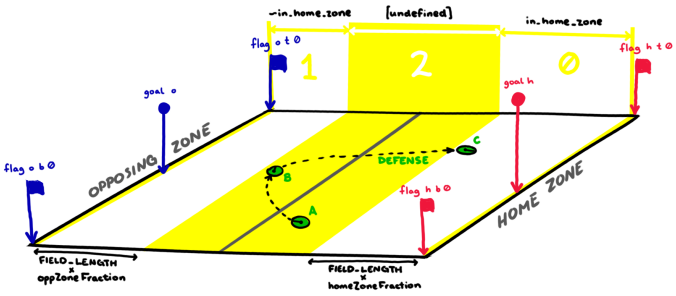

Fig. 4. A visualization of the zone belief interpretations performed by the defensive agent.

Note that each defender has a designated flag assigned to them, at an angle of $\pm$ 40 degrees from the center of the goals. Upon returning to their home zone, the defenders align themselves with their designated flag. This ensures that they will return at a specific angle, and thus be spread out.

Finally, the defenders need to cooperate properly with the goalie. As such, the defender's third goal is to ensure that they only go for the ball when it is not in the possession of the goalie. To achieve this, the defender uses its distance and relative angle with the goalie and the ball to estimate the distance between the goalie and the ball. It then checks this distance against a threshold (the same threshold used by the goalie to catch the ball) to see if the goalie is going to attempt

to catch the ball. An example of this is shown in Figure 5. The variables in the figure are defined as follows:

- `d_ball`: defender's distance with the ball.
- `d_goalie`: defender's distance with the goalie.
- `d_goalie_to_ball`: goalie's distance with the ball.
- `ball_angle`: defender's angle with the ball.
- `goalie_angle`: defender's angle with the goalie.

Using *cosine's law*, `d_goalie_to_ball` can be calculated as follows:

$$\text{angle\_in\_between} = \text{goalie\_angle} - \text{ball\_angle}$$

$$\text{d\_goalie\_to\_ball} = \sqrt{d_{\text{ball}}^2 + d_{\text{goalie}}^2 - 2 \cdot d_{\text{ball}} \cdot d_{\text{goalie}} \cdot \cos(\text{angle\_in\_between})}$$

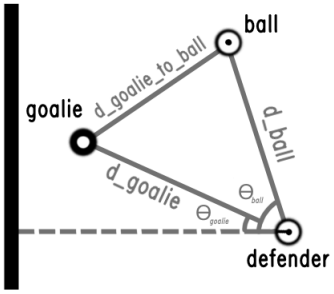

Fig. 5. A visualization of how the defender estimates the distance between the goalie and the ball.

Once the defender has concluded that the goalie has the ball, it checks to see if it is too close to the goalie. If it considers itself to be too close (less than 10 m) to the goalie, it heads back toward the center of the field at the same angle that it uses for its designated flag. This ensures that the defenders will be spread out if the goalie catches the ball and performs a free kick. Once the defenders are far from the goalie, they will wait until the goalie has released possession of the ball.

C. Attacker

The offensive player agent (defined by `attacker.asl`) can be considered an inverse of the defender agent, where similar beliefs are interpreted to keep the player in the oppositional zone. Whereas the defense agents are soft-bound to play in the home zone, the offensive players will relocate to forward positions, defined by aligning themselves with a specified offset and running to the area of the field where the `in_home_zone` belief is added to the `JASON` agent. This behavior is only enacted when the `JASON` framework believes `not(ball_seen)`, and the offensive player can still play defense within a certain range, if necessary. In order to prevent over-aggression by means of triggering the offside penalty, the ability to detect offside conditions was integral to the pacing of the offensive agent. To this end, a triangulation algorithm was implemented to update the agent's beliefs (6).

The agent will iterate over all visible enemy players, then compare them with the perceived `h` distance of the observed ball. If the ball is located behind all enemies (excluding the opposing Goalie), the agent will run at a reduced rate instead

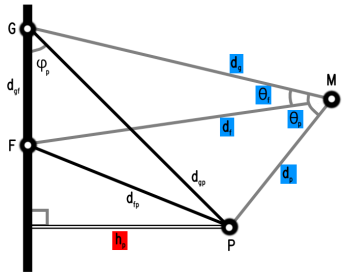

Fig. 6. Triangulation scheme used to detect offside conditions. Law of Cosines are applied to retrieve the distance from the opposition line h_p

of a full sprint, usually ensuring the ball will be returned to the possession of the opposition, avoiding an offside foul. Note that the previously calculated zone beliefs are applied to the detection of offsides, as they can only occur in the oppositional zone.

III. Agent Software Design

When implementing the BDI logic using `JASON`, careful management of active beliefs and their catchment was essential to maintaining defined agent behavior. In order to orchestrate the goal transitions of the agent, we modelled a hub-and-spoke flow to triggering events, centralizing the decision-making on the `!wait` event, with "spokes" being extant subroutine-style plans that handle a certain aspect of agent behavior. These branches tend to serve `!wait` once the desired behavior is complete. The goalie agent, due to advanced positioning mechanics, has more extant branching involved, but the return-to-idle behavior remains consistent.

IV. Running the JasonBDI Soccer Agent

If you are on Windows simply double-click on the `TeamStart.bat` file to run two teams with the `JASON` player types described in this report. Similarly, if you are on a `UNIX` machine you can run the shell script using:

```
$ ./TeamStart
```

For the initialization of a single team (5 players), the batch script `HalfTeamStart.bat` Windows and `HalfTeamStart.sh` can be run instead. These scripts are designed to allow for running the players against another RoboCup Team.