Symbolic AI Group Project

Group Members:

1. **Student ID:** 6638014021 **Name:** Kantinan Sriratbunterng
2. **Student ID:** 6638076921 **Name:** Thienkwan Boonchai
3. **Student ID:** 6638079821 **Name:** Thanatat Dheravijaranayankul
4. **Student ID:** 6638199421 **Name:** Mohamed Farzan Mohamed Sathik
5. **Student ID:** 6638248521 **Name:** Siravich Buddhaunchalee

---

In this project, our group was assigned to implement an elementary RoboCup simulation using Symbolic AI and logic programming with Prolog. The task involved designing a rule-based system to control players in a dynamic soccer environment, integrating key AI concepts such as knowledge representation, logical reasoning, decision-making, and planning, along with using Prolog in symbolic computation to develop agents capable of sensing, reasoning, and taking strategic actions based on predefined rules. Therefore, this report aims to detail our approach in developing the system, covering symbolic representations, strategies, and the rationale behind our design and implementation decisions. Moreover, it will evaluate our program's performance, identifying its strengths and areas for improvement.

1.0 Initial Planning and Research

Our group began by reviewing the instructions provided by Professor Miri, as well as the PDF files of Michael Floyd's RoboCup presentations from 2008 and 2012. These files offered a detailed overview of the RoboCup challenge and helped inform our approach to the simulation. However, the documents mostly focused on a Sub-Symbolic approach, while this project required us to develop a small-scale simulation using Prolog and apply the understanding of Symbolic AI that we learned in class. Therefore, we decided to research further on similar projects that use Prolog, specifically focusing on RoboCup simulations, and examined example code from GitHub to gain a deeper understanding of how the system functions in action. Additionally, we observed RoboCup simulation videos on YouTube, which provided practical demonstrations that helped us gain a clearer picture of how each agent senses the environment, makes decisions, and executes actions within the game. By combining our understanding of the provided documents with this supplementary research, we aimed to build a solid structure for designing and implementing our Prolog-based solution.

In the process of planning the design, we first focused on the key elements required for our program, such as defining the game environment, including the field, ball, and players. We then proceeded to implement agent behaviors using goal-based agents and defined essential facts and rules in Prolog for different player roles, such as kicking the ball, catching the ball, and moving toward the goal. The reason a goal-based agent is well-suited for this challenge is that RoboCup simulations are dynamic and goal-oriented. Agents must react and make decisions based on specific objectives, such as scoring a goal, defending, or passing the ball, which makes goal-based agents a suitable fit for handling the constantly changing environment and adapting strategies throughout the game.

After some initial planning and discussion, we used the sample code structure provided as our base and filled in some missing parts (without adding any new predicates yet). We implemented a rough version to run our first test and observe the results. However, when we executed the code for the first time, the simulation encountered multiple errors due to incomplete or missing predicates and logic. Despite this, we were still able to run the first few rounds, which helped us gain a better understanding of how the simulation works frame by frame, where each round acts as a single frame. In each frame, the system updates the game's state by determining the actions of all agents based on their current positions and predefined rules. The game continuously updates the states in each frame until the round ends or a specific objective is achieved.

Meanwhile, one major challenge we encountered during testing was the difficulty of debugging using only coordinate printouts, which made it hard to track how each agent behaved due to the lack of a visual representation. To address this, we decided to divide our group into two sub-teams **1) Visualization:** One team focused on developing a visualization tool to make debugging easier and improve testing efficiency, and **2) Logic Code:** The other team handled the core program code and logic implementation. This allowed us to organize tasks more efficiently.

2.0 Visualization

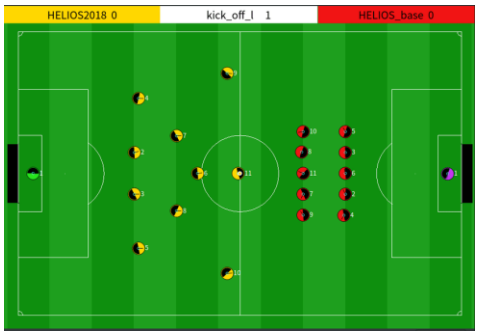

Figure 1: A visual representation of the RoboCup simulation from a YouTube video.

In this part, our visualization team attempted to draw elements such as the field, ball, and players similar to those in Figure 1. However, Prolog itself does not have built-in libraries for creating visualizations, and we were not familiar with any external tools that could generate graphical representations. Therefore, we began searching online for available tools that could assist with visualization. Since we found limited examples of visualization in Prolog, we used AI models like ChatGPT to find more recommendations on suitable tools. Based on the suggestions provided, we chose XPCE, which is a GUI toolkit that enables the creation of graphical elements such as windows, shapes, and animations. We selected XPCE because it had the most available examples and demonstration videos from other game projects on YouTube.

As shown in Figure 2, we attempted to draw the soccer field by setting its dimensions to 1000 x 500 pixels. First, we created a 1000 x 500-pixel window using the XPCE library. We then drew boundary lines slightly smaller than the full field to create a clear border and placed a center circle in the middle. Next, we added goalposts at both ends of the field and placed players as small colored circles. However, there was an issue with the coordinate system, as the ball and players were based on a smaller scale in our backend code. To match the intended display size, we needed to scale up all positions by a factor of 10 when converting our logic to pixel-based rendering. For example, a ball defined at (50, 25) in the backend was displayed at (500, 250) in the graphical interface. By applying this scaling factor, all elements in the Prolog logic were correctly aligned in the visual representation. Additionally, we set the field color to dark green and changed all boundary lines to white for better visibility. We also added decorative elements, such as the midfield line, which is shown in Figure 3, to make the field look more similar to the soccer field example shown in Figure 1. After setting up the field, we attempted to add more players and implemented a move_ball predicate to test position updates. However, we encountered significant lag in the visual rendering when updating coordinates, which caused a big challenge for our visualization team to resolve.

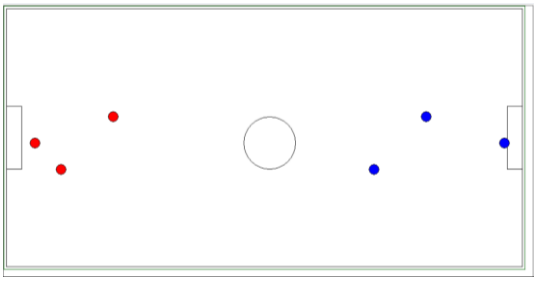

Figure 2: Initial draft of the soccer field visualization.

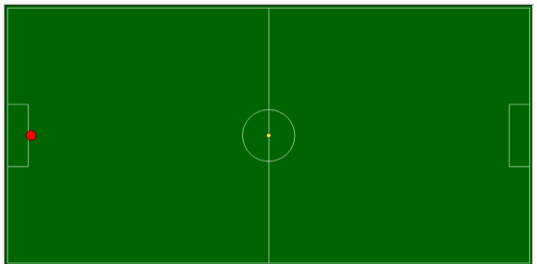

Figure 3: Second draft of the soccer field visualization.

At that point, we still could not find a way to solve the problem, so we came up with a temporary plan to create a visual using text-based representation, similar to the Tic-Tac-Toe and Sudoku examples we learned in class.

In Figure 4, this is the text-based visual representation that was used to display and debug the code in the logic part. The ball is represented by the capital letter "O", while Team 1's players are denoted by the capital letters "G" for goalkeeper, "F" for forward, and "D" for defender. The same thing applies to Team 2's players, but with lowercase letters.

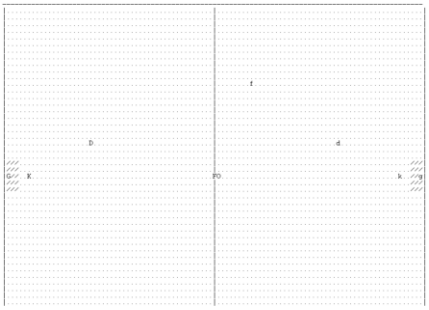

Figure 4: Text-based visual representation of the soccer field.

3.0 Logic and Gameplay System

Creating a full RoboCup simulation is highly complex due to its advanced logic and lots of components. Our project, however, is a small-scale simulation that focuses on essential mechanics like player movement, ball interactions, and role-based behaviors. Instead of replicating every small detail, we used simple movement patterns and basic player behaviors to make the system easier to manage. With the flexibility provided, our group defined our own rules and gameplay mechanics for our simulation, keeping only the necessary elements while ensuring our implementation followed the given instructions.

After combining the text-based visual with the rough version of the logic code mentioned earlier, we identified several issues. For example, the move_toward_ball predicate, which is currently applied to all players, causes every player to move toward the ball simultaneously, resulting in a large cluster of players at a single point on the soccer field map. To address this issue, we created a list of rules and constraints that outline how each role should function and the specific predicates we needed to write for each role on the team, as shown below:

Agents: 1) Forward 2) Defender 3) Goalkeeper

Forward:
Goal: Move toward ball, kick (scoring)
State Space:
- Free ball: Run toward ball.
- Ball with opponent: Run toward ball (but hold position if too far).
- Ball with forward: Dribble toward goal. *If within scoring distance (e.g., 10 units), kick for goal.*

Defender:
Goal: Steal ball from opponent, pass ball.
State Space:
- Free ball: Run toward ball (but hold position if too far).
- Ball with opponent: Run toward ball.
- Ball with defender: Dribble toward forward and pass ball to them.

Goalkeeper:
Goal: Catch ball, pass ball.
State Space:
- The goalkeeper always moves up and down along the Y-axis.
- Ball inside goal area: Move toward ball (but stay within the goal area).
- When ball is outside goal area: Stay in position.

Game Rules:

Starting:
- Team 1's forward always starts in the center with the ball.

Goal & Reset:
- When a team scores, the game pauses for a short break, then resets with the opponent's forward positioned in the center.
- If the goalkeeper catches the ball, it will pass to the nearest forward player within its range.

Game Duration:
- The match lasts for 1 minute 30 seconds with only 1 round.

Winning Condition:
- The team with the highest score at the end wins.
- If there is a tie, the match ends in a draw.

Stamina System:
- **Move:** -1 stamina per move.
- **Stay still:** +5 stamina (rest).
- **Pass ball:** -2 stamina.
- **Kick ball (scoring):** -5 stamina.
- **Catch ball:** -5 stamina.

Figure 5: A list of rules and constraints of our game-play.

Design Methodology:

In this part, we mainly focus on the design of predicates along with explaining the important predicates used in our game and approach the problems we faced and how we tackled them, which we explained separately into Main predicates and Helper predicates for the ease of understanding and visualising the gameplay.

Main Predicates:

- move_towards_ball: As mentioned earlier in the report about the problem regarding move_towards_ball predicate applied to every player, we decided to add a condition to the predicate to split its functionality to forward and the defender. The functionality of the forward and defender vary, and they are briefly mentioned in Figure 5.

- dribble: Typically, to simplify the code, the dribble action is designed so that the ball gets pushed toward the goal as it is received. However, we introduced a custom dribble predicate to enhance the game's originality and provide a more immersive experience. Our dribble mechanic ensures that the ball stays with the player and moves along with them.

- tackle: Since we introduced the dribble predicate, we added a tackle predicate to be able to steal the ball leading to a fair gameplay. To achieve realistic play, we set the tackle succession chance as random (50%). Moreover, we encountered some issues when tackling the ball, the tackling player gets tackled back immediately as the players are near each other, leading to a loop of tackling and re-tackling. To overcome this, we set the tackling player to be able to sprint (dribble) 6 units to avoid getting re-tackled. If the player with the ball encounters tackle but the tackle fails, the player with the ball will sprint (dribble) 4 units to avoid getting tackled again.

- pass_ball_def: As mentioned in Figure 5, since the defender cannot score a goal, we assign it to always pass the ball to its forward teammate. But in order to promote a realistic gameplay, we allowed it to dribble the ball towards the forward before passing it. We also set the passing ball action to be random (near the forward from top or bottom), replicating real game play. This allows us to produce a variety of actions in a game play.

- kick_ball: We set on only allowing the forward to be able to kick the ball as indicated in Figure 5. When the forward is within the range near the goal position, it will then automatically kick the ball to the goal, which we instantiated to random to the coordinates of the goalpost in Y-direction for the direction of the ball to the goal.

- goalkeeper_move: During the opposing forward with ball is near the goal position, the goalkeeper will move forward within the goalkeeper outline to catch the ball.

- pass_ball_gk: After the goalkeeper is able to catch the ball, the goalkeeper will throw the ball ahead of its teammate forward (around 3 units ahead).

The predicates mentioned above are only some examples of the main predicates that have been modified to our gameplay and we would like to note that our game has more generic predicates used than ones mentioned. Plus, we used helper predicates to help with directing to the main predicate to make the code easier to read and debug while needed. One of the examples was the helper predicate check_bounds, where we ensured that the play doesn't occur outside the field, which is used in several of the main predicates. There is more helper predicates used in our code, a few of them are mentioned below:

Helper Predicates:

- **kicked**: indicates which team that kicked the ball.
- **catchable/uncatchable**: determines if the goalkeeper is near/far enough to catch the ball.
- **ball_caught_by**: indicates which team's goalkeeper has the ball.
- **distance:** calculates the distance between point A and point B.
- **check_bounds:** checks/assures that the play is within the field size.
- **ball_in_range:** checks if the player is near the ball for possession.
- **find_nearest_teammate:** locates the teammate nearest to the ball.

Overall, we use " simulation_round " predicate which combines and reflects all the functionalities of predicates used in this game, start with check goal, then prioritize action associate with goalkeeper, check ball status (to assign status to nearest player), forward and defender action, where forward and defender choose one action per round (not include tackle), and lastly each team can tackle one time per round (choose forward or defender to tackle), show ball position, display, and update score.

4.0 Implementation and Finalization

To optimize performance, we restructured draw_players, draw_ball, and draw_field to create objects rather than redrawing them each frame, reducing lag. We added update_player, update_ball, and update_score to efficiently handle position updates and score changes. Additionally, we included labels for each player that display their role and stamina, providing more detailed information about each player during the game. We also added a box at the bottom to display the current score. The final visualization is shown in Figure 6.

Sample Video of How Our Simulation Works

Figure 6: The final visualization.

After completing the logic and visualization, we connected the two code sections and made necessary adjustments. We also added show_goal_message and show_end_message to display goal notifications and match results. A pause after the goal notifications was implemented for 3 seconds to allow smoother transitions between rounds, improving the flow of the simulation.

5.0 Conclusion and Recommendations

Our Prolog Robocup Simulation has come out pretty well as it ticks all of the boxes of requirements mentioned in the instruction document. It reflects key symbolic AI principles in various parts of the game. Stamina tracking, role-based behaviors, ball control, and decision-making are few of the immersive features our game implies of the principles learned in class.

One of the main strengths is our implementation of tackling mechanics. Instead of relying solely on passive approaches like passing and following the ball, our simulation enables defenders to actively steal the ball, reflecting real-world soccer dynamics. Another major strength is the visualization system, it clearly displays the role and stamina of each player above their position on the field. This design helps users track the game state in real time and better understand the logic behind each player's behavior. The simulation also includes realistic game mechanics such as energy consumption, dynamic kicking, goal detection with score updates, goalkeepers' actions, turn-based randomness, and post-goal resets, while maintaining an efficient and readable structure.

On the other hand, there are areas for improvement. We observed occasional visual glitches, particularly where the player labels incorrectly display score values instead of roles or stamina. Thus, we tried to fix the bug and check the variable if it has been assigned correctly, but the result was inconsistent. While the current 3 players per team set up is enough, increasing the number of players could enhance diversity of gameplay. For further recommendation, we suggest adding more advanced decision-making logic such as dodge behavior or formation changes. These enhancements would elevate the realism and experience of the game.