A High Level Language Based on Rules for the RoboCup Soccer Simulator

Jose I. Nunez-Varela

Abstract— **The RoboCup Soccer Server is an excellent framework where students can learn and understand more about artificial intelligence (A.I.) application problems. However, getting to work with the server is not a straightforward task, they are required to know specific details about the server, which is time consuming and complex. This paper describes the development of a high level language based on rules for the Soccer Server. This language provides a full set of conditions and commands to create rules that will determine the behavior of a player, in an easy and quick manner. Thus, students can focus on problems such as player's behaviors, strategies, coordination, etc.**

Index Terms— **A.I. Education, Multi-Agent Systems, Rule-Based Systems, Soccer Simulation.**

I. Introduction

ROBOCUP is a research and educational initiative. Its objective is to foster the A.I. and intelligent robotics research by providing a standard problem [4]. There has been a continuous effort to increase the motivation, participation and interest of students in science and technology.

The RoboCup soccer server provides an excellent platform for students to learn and apply A.I. techniques. It does not require the use of real robots, so students or universities who do not have the resources to participate in other leagues are not exempt for participating actively in the RoboCup domain.

A. RoboCup Soccer Server

The soccer server is a powerful tool that simulates games of soccer in the best possible way. The simulation is executed in a client/server style via UDP/IP sockets. A soccer team will normally consist on 11 players (10 players and 1 goalie), and a coach. Each player and the coach are clients that would connect to the server. Also, a monitor is needed to visualize the simulation, which is another client too. Once all players are connected to the server, the server will send sensory information to every player. This information will help the player to make a decision on what action to perform next. This action (command) is then communicated to the server, which will simulate it on behalf of the player.

Manuscript received August 28, 2007. This work was conducted at the Computer Science Department, University of Pittsburgh.

J. Nunez-Varela was a M.S. student at the Computer Science Department, University of Pittsburgh, Pittsburgh, PA 15260 USA. He is now with the School of Engineering, Universidad Autonoma de San Luis Potosi, San Luis Potosi, 62490 Mexico (e-mail: jnunez@uaslp.mx).

The soccer server sends three types of sensory information at different intervals in time. Furthermore, the players should send their commands at certain intervals also, so the server could simulate them correctly. The consequence is that users should be familiar with real-time systems, and also, they should code routines to handle this synchronization problem.

The sensory information contains only local information for every player. So in order to make good decisions, a player must have a global perspective of what is going on inside the soccer field. For instance, it is desirable to know our position and the positions of our teammates, opponents, the ball, etc. Therefore, users should build a software system capable of creating a global representation of the game.

Players must send commands to the server in order to do something. These commands are primitive and must contain local information also. Good plays are coded with the combination of these commands. However, users must calculate the parameters needed to perform each command. For example, the *kick* command uses two parameters: *power* and (local) *angle*; then we must first calculate these two parameters in order to perform that kick.

We can see that the soccer server is a robust and excellent simulator system [3]. Unfortunately it might be complex and difficult to understand and utilize for some people. Especially if the simulator is intended to be used as an educational project. For example during an A.I. course where students do not have enough time to familiarize with its functionality. Also, students might not be familiar with sockets programming and real-time systems at all. Therefore, the development and implementation of a full soccer team requires a lot of effort and time that could sometimes be impossible to spend. This might lead to the total loss of interest not only from the students but also from the professor.

B. Objectives of Our Project

Based on the idea of using the soccer server as an educational project we have developed and implemented a high level language based on rules that could be easy to use and understand. Trying to eliminate the aforementioned problems and allowing the students to create a full team of players in a straightforward manner with little knowledge about the server and its complexities, thus saving time and effort.

The language provides a complete set of directives, in the form of conditions and commands. These are used to create rules to specify the behavior of a player, and ultimately the

behavior of a full team. This allows the students to quickly explore the creation of strategies, team coordination and planning, not worrying about such things as how to send a message to the server.

Since the main objective was to create a language with a large and functional set of conditions and commands, there was no point in creating from scratch a working soccer system. Therefore, we chose the UvA Trilearn robotic soccer simulation team [1] as our platform to build on top of their system our language. To this end, we modified certain parts of their code and added some other functionalities that were not available in the original code.

It has already been planned to use the language as part of an undergraduate A.I. course. The intention is to use the soccer simulator so that students can create their own soccer team and then organize a tournament at the end of the term.

II. A High Level Language

While the functionality of the soccer server is somehow easy to comprehend, the construction of just one player is complex and time consuming. There is a large number of considerations that must be taken into account in order to build a software system that could work with the soccer server. Hence, the availability of a high level language easy to understand and use, would result in an excellent way of allowing a large number of users the possibility of creating their own teams in a straightforward manner. Requiring a small amount of knowledge about how the soccer server works.

A. UvA Trilearn Team

Since the establishment of the RoboCup competition several universities and research teams have spent a lot of time and effort in developing good software players. And in an effort to expand the interest for this problem they have been releasing their source code so that other people could use it and work on top of it. Avoiding the overhead of developing a new software system.

Since our interest was on the high level language, we decided to work on top of an existing system. We found that the UvA Trilearn simulation code [5] was our best choice. One of their main objectives was to have a good base code that could be used in the following years after its creation in 2001. Thus, a lot of effort was put into solving certain low-level aspects of the server, such as the synchronization problem and the implementation to represent the player's world model. It follows an object-oriented approach, so it is easy to modify and update with functionalities that were not planned originally. Their code is also very well structured and documented. And it was done considering the most recent version of the soccer server (version 10).

It can be argued that students could just work on some free source code such as the UvA Trilearn system. However, it still requires time to understand and get familiar with the server and their code. Also, it requires good programming knowledge in order to make changes and create their own players.

B. Rules

Rules are a natural way to describe situations where conditions must be satisfied in order to perform some action. A rule is of the form:

If (*conditions*) **then** *perform some action*.

Rules can easily be used to code the behavior of a soccer player. For example we could define the rule:

If *I can score a goal* **then** *kick the ball to the goal*.

Rules allow to create reactive players, in the sense that given the current circumstances the player will decide which action to perform. Rules are easy to read and understand, especially if they can be written in natural language such as English. And thus, rules are also easy to create and modify. For these reasons we decided to develop a language based on rules. The specific syntax for our language is the following:

if *(condition(s))*
then *command(s);*
end

We can combine conditions using logical operators: *(&&)* for AND-operation, *(||)* for OR-operation and *(!)* for negation; or relational operators: *(<)* is greater than, *(>)* is less than, *(<=)* is greater-equal to, *(>=)* is less-equal to, *(==)* for is equal, and *(!=)* for is different.

C. Player Modes

We have mentioned that the use of rules produces reactive behaviors. But in certain circumstances that kind of behavior might not be useful. Thus, a planned strategy could work better. Therefore, we have incorporated the use of "player modes". Modes allow the user to group rules and to activate them later in situations they may find appropriate. For example, we could plan a certain strategy in the case of a corner kick or free kick. We could also change the player's behavior if certain condition is satisfied. For example, if there are no opponents a defense could change its mode to attacker to try to score a goal even if it is not its primarily function. Once it notices that it has been blocked by some opponent it will return to its normal behavior. A mode is specified as follows:

mode *name_of_mode*
:
rules
:
endmode

When we change to some mode we could also specify the frequency to which we would like to use that mode. This means that a player may or may not change its mode, adding a surprise factor to its behavior.

III. Implementation

Each condition and command that we defined for our language has to be translated into C++ code and be added into the UvA Trilearn system to create the executable file. Therefore, modifications and additions to the UvA Trilearn code were needed in order to allow a clean and organized

translation. Furthermore, modifications were also needed to increase the functionalities of their code, to allow the definition of more conditions and commands for our rules.

As mentioned before, the UvA Trilearn code follows an object-oriented approach. One of our main additions was a class containing all the methods needed in order to perform every condition and command. We consider this class to be a good addition to their code. Anyone familiar with their code could create players easily using those methods.

Other classes were modified to add more functionality. For instance, the class defining the formations of the players was modified in order to have better formation stances among players. A class defining areas and zones inside the soccer field was added. This class is used to tell the players where to go or to specify their position. We defined ten areas inside the field and each area is divided into nine zones. Thus, a player can easily be positioned in any part of the field.

A. Constants and Parameters

Before we start describing the conditions and commands available in our language, we will briefly describe some of the constants and parameters that should be used within our language. These will let us specify situations, objects, positions, directions, values, etc. We have classified these constants and parameters as follows.

1) Play modes: These constants are used to refer to some specific mode of play during the game. The referee specifies these modes. For example: *BEFORE_KICK_OFF* tells us that the game has not started yet. Other examples include: *FREE_KICK_US, OFFSIDE_THEM, PENALTY_US, CORNER_KICK_THEM*, etc.

2) Objects: These constants refer to players and the ball. For example: *TEAMMATE_1* specifies the reference to our first player. Other examples include: *BALL, OPPONENTS, OPPONENT_6, ATTACKER, GOALKEPPER*, etc.

3) Directions: For example: *EAST* specifies the east direction. Directions are relative to coordinates of the field from the player's perspective. This means that they do not change, even if teams swap sides in the halftime of the game. Constants for distance are also specified, such as *FAR* and *NEAR*.

4) Positions: These constants refer to specific areas and zones inside the field. For example: *OPP_PENALTY* determines the opponent's penalty area. Other examples include: *OWN_CENTER*, *ANY_ZONE*, *BOTTOM_ZONE*, *OPP_TOP*, etc.

5) Actions: These constants are used for actions such as: Passing, kicking, dribbling, tackling and marking. For example: A player may try to kick the ball into the goal at the *TOPCORNER, CENTER, BOTTOMCORNER* or *RANDOM*.

6) Stamina: We use these constants to check the player's tiredness during the game. The stamina level may be *GOOD, AVERAGE, BAD, VERY_BAD, TERRIBLE*.

7) Parameters: We can specify Boolean values and numeric values directly to evaluate some condition. Boolean values are *TRUE* and *FALSE*. And numeric values can be natural or real numbers.

B. Conditions

Conditions allow the user to verify situations or events during the game. The number of rules that can be created is proportional to the number of conditions available. Therefore, our intention was to create a full set of conditions. Trying to cover every possible situation in a soccer game. We have defined around one hundred conditions. Obviously, they can be combined with relational or logical connectors to form more conditions. We have classified the conditions as follows.

1) Formations: This set of conditions are used to determine the information about the team formation. For example: *isInFormation* determines whether a certain player is on its defined position, as the current formation specifies.

2) Areas/Zones: These conditions are designed to work with the areas and zones inside the field. For example: *isClearAtArea* determines whether or not there are teammates or opponents inside a certain area and/or zone.

3) Stamina: These conditions are defined to let the player know its current and future condition. For example: *isMyCondition* determines whether the condition of the player is the same as the given parameter.

4) Communication: This set of conditions are used for coordination and planning. For example: *isHeardMessage* determines whether the player heard a particular message.

5) Player: We use this set of conditions to know certain information about the player's perspective. For example: *amICloseTo* determines whether our player is close to some object. Another example: *isHeadingToMe* determines whether the ball, a teammate or an opponent is moving towards the player's position.

6) Game: This set of conditions have to do with the status and some situations in the game, and about the status of the ball. For example: *isBallKickable* determines whether the player is capable of kicking the ball. Another example: *isInOffside* determines whether some teammate is currently in an offside position.

7) Direction: These conditions require the specification of the direction in which they are to be verified. For example: *numPlayersAroundMe* returns the number of teammates or opponents inside a circle with center at the current player's position and with some radius.

8) Pass: These conditions help the player to decide whether or not is good to pass the ball. For example: *canIPassToClosest* determines whether or not it is possible to pass the ball to the closest teammate, since it is probable we do not know its exact position.

9) Tackle: These conditions allow the player to decide whether is good to perform a tackle or to verify if somebody is going to tackle us. For example: *getProbToLoseBall* returns the numeric probability of being tackled by some opponent.

10) Server: In order to make our language more robust, it allows the user to have access to the values the soccer server handles directly. Although all the high level conditions provide enough information to make decisions, in certain situations it

might be desirable to get detailed information in order to make better decisions. For example: *serverSpeed* returns the current speed of the player (value between 0 and 1.2). Of course, the use of this set of conditions requires knowledge about the soccer server.

C. Commands

Commands are the actions we want the player to perform when some conditions are satisfied. As with the conditions, we have defined what we think is a large number of commands. A player might perform almost sixty commands during the game. We provide the following classification.

1) Formations/Areas: These commands are designed to work with the player's formation and its position. For example: *moveToArea* makes the player to move to a certain area and/or zone. Once the player reaches the area it will stop moving.

2) Ball: These commands are related with the use of the ball during the game. For example: *kickToGoal* makes the player to kick the ball to some desired position and speed, to the opponent's goal. *interceptBall* makes the player to move toward the ball as fast as possible in order to intercept the ball. *passToClosest* allows the player to pass the ball to the closest teammate.

3) Player: This set of commands control the player's body. For example: *turnBodyTo* makes the player to turn its body toward some object.

4) Goalie: This set of commands are used exclusively by the goalie. For example: *catchBall* allows the goalie to catch the ball if it is within the "catchable" area.

5) Communication: These commands are used to communicate with other teammates for coordination and planning. For example: *sayMessage* makes the player to say some message.

6) Game: This set of commands are used for defending, searching and moving. For example: *searchFor* makes the player to search for some object in the field. Another example: *markTo* allows the player to mark some specific opponent.

7) Server: As with conditions, we decided to provide commands that the server understands directly. We may want in some situations to have direct control of what the player should do. For example: *serverDash* makes the player to dash in the direction of its body with the supplied power (0 -100). Users should know what kind of parameters and ranges the server accepts in order to use these commands.

IV. Functionality

Basically, the user should write the rules for the players into a text file. Then that file is compiled and a C++ file is generated, which will be linked together with the UvA Trilearn code.

A. Compiling the Rules

Rules should be written into a predefined text file. In fact, we have eight available files, each one with the name of a different type of player, according to those found on the UvA Trilearn team. We could expect for example that rules specifying the behavior for the goalie should be written into the *goalie.rules* file, and so on. However, it is important to point out that these files are only intended to provide an organized manner for describing the players. It does not mean that the *goalie.rules* file must have the rules for the goalie. After all, rules define the behavior of the player, not the name of the file. Furthermore, it is not necessary for the user to use all the available files. The user could choose to write rules for all the players in just one file. This is possible if we use the changing of modes explained earlier.

The compiler used to parse the rules was programmed in C++ for Unix. Fig. 1 illustrates a schematic view of the compilation process. The compiler analyzes the rules specified in the files for each player, if everything was correct it will generate a file named *PlayerTeams.cpp* which will contain the C++ code which will be linked along with the UvA Trilearn system.

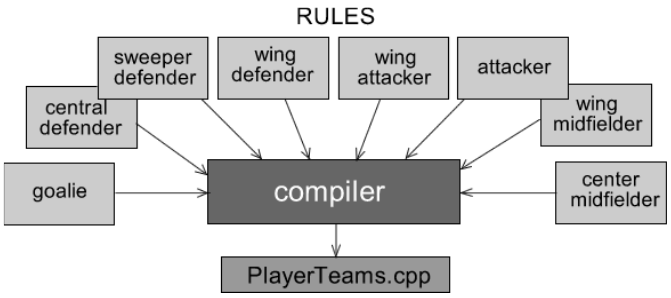

Fig. 1. Schematic view of the compilation process.

B. Examples

It is important to know that the order of the rules determines their priority. We can think of rules as being inside an infinite loop. The program will check from the beginning of the loop for the satisfaction of some rule. Once a rule is satisfied the command(s) will be executed and the loop will start all over again from the beginning. This kind of logic is similar to the form humans play. If we are running to some position but suddenly we can be able to kick the ball, we are expected to do so, and not to keep running. Thus, we set priorities in our actions. To demonstrate how to create rules and how easy is to define players, let's review some examples.

1) Kiddie Soccer: This is a common term used to refer to the way small kids play soccer. At that age they do not have a good sense of coordination and teamwork, and thus everybody try to kick the ball and to follow the ball through all the field. We can build such kind of player with the following rules:

```
if (isBallKickable)
then kickToGoal (POS_RANDOM, MAX_SPEED);
end
if (!isKnownPosition (BALL))
then searchFor (BALL);
end
if (isKnownPosition (OBJECT_BALL) &&
   (isMyCondition (GOOD) || isMyCondition (AVERAGE)))
then moveToObject (BALL);
```

```
end
```

This example is also good to show that the ordering of the rules is important. If we write the first rule at the end, the player will never kick the ball, it will only chase the ball. Unless it gets tired and the ball is in a kickable distance.

2) Attacker: This example shows a basic behavior expected for an attacker. The first rule makes the player to kick the ball whenever it is possible inside the opponent penalty area. It will kick the ball to the goal randomly. The second rule applies when there are no opponents in the direction of the penalty area. The third rule will try to approach to some area where the ball can be kicked to the goal. The fourth rule makes the player to try to get control of the ball if the player is closest to the ball with respect to its teammates. The fifth rule makes the player to go to its position if it is idle. And finally, the sixth rule allows the player to know the position of the ball.

```
if (isBallKickable &&
   amIInArea (OPP_PENALTY,ANY_ZONE))
then kickToGoal (RANDOM, MAX);
end
if (isBallKickable &&
   isRectClearAtDir (OPPONENTS, EAST, FAR))
then kickToGoal (RANDOM, MAX);
end
if (isBallKickable)
then dribbleToDir (EAST, SLOW);
end
if (isKnownPosition (BALL) &&
   amIClosestTo (BALL, TEAMMATES))
then interceptBall;
end
if (!amIInArea (OPP_CENTERTOP, CENTER_ZONE))
then moveToArea (OPP_CENTERTOP, CENTER_ZONE);
end
if (!isKnownPosition (BALL))
then searchFor (BALL);
end
```

V. Conclusions and Future Work

This paper described the development and implementation of a high level language based on rules for the RoboCup soccer server. Its main objective is to eliminate the inherent complexities found on the soccer server. Thus, allowing the users to create soccer players in a straightforward manner with little knowledge of the soccer server. This is desirable for certain situations. Here we talked about the possibility of using the soccer server as an educational project. Students could work on the server to learn about multi-agent systems, coordination, planning, decision making, among other things.

We believe that RoboCup competitions are attractive for a large number of students. However, if they find that using the soccer server requires too much effort, they could lost interest very quickly. Hence, by using a language that allows them to create players easy and fast, it will help them to maintain their interest. Furthermore, involving the students faster into the real problem might motivate them to want to know more about the server, learn how it really works and how a player can be enhanced.

There are some upgrades we are planning to implement in our language. First of all, increase the set of conditions and commands to comprise more situations and to make the rules more flexible. Also, the whole compilation process could be done in more user-friendly manner. And the most important thing is to release our code so that it can be downloaded and used by anyone who might be interested. We have not done it because there is no user manual that can be attached to the code yet. So we have planned to complete the manual and set up a website to upload the project.

Further work might also include implementing a high level language for the 3D simulation server. And also, implementing a high level language for coaches.

As we mentioned before, we are planning to use the soccer server as part of an undergraduate A.I. course. We hope that the students will take advantage of our implementation. And also, we expect that the interest in A.I. and robotics research can grow with this kind of projects. Since it is also our goal to form a group of students with the desire to participate in some RoboCup competition in the future.

Acknowledgment

The author wish to thank Dr. Milos Hauskrecht for his helpful advice and assistance throughout this project.

References

[1] R. de Boer, J. R. Kok, “The incremental development of a synthetic multi-agent system: the UvA Trilearn 2001 robotic soccer simulation team”, M.S. thesis, University of Amsterdam, The Netherlands, 2002.

[2] R. de Boer, J. R. Kok, F. Groen, “UvA Trilearn 2001 team description”, *RoboCup 2001: Robot Soccer World Cup V. Lecture Notes in Computer Science*, vol. 2377, Springer-Verlag, pp. 551–554, 2002.

[3] E. Foroughi, F. Heintz, S. Kapetanakis, K. Kostiadis, J. Kummeneje, I. Noda, O. Obst, P. Riley, T. Steffens. (2001). RoboCup soccer server user manual: for soccer server version 7.06 and later. Available: http://sourceforge.net/projects/sserver

[4] H. Kitano, M. Asada, Y. Kuniyoshi, I. Noda, E. Osawa, “RoboCup: the robot world cup initiative”. *Proc. of The First International Conference on Autonomous Agent*, Marina del Rey, California, United States, 1997, pp. 340-347.

[5] J. Kok, R. de Boer. (2003). The Official UvA Trilearn Source Code. University of Amsterdam. Available: http://www.science.uva.nl/~jellekok/robocup

[6] I. Noda, H. Matsubara, K. Hiraiki, I. Frank, “Soccer server: a tool for research on multiagent systems”. *Applied Artificial Intelligence*, vol. 12 (2–3) pp. 233–250, 1998.

[7] The RoboCup Federation: the official RoboCup website. Available: http://www.robocup.org/