Research on ROBOCUP Soccer Simulator2D COMP400

Pedro Khalil Jacob

Under the Supervision of Professor Joseph Vybihal

Department of Computer Science

McGill University

April 23, 2022

I. Introduction

Soccer is the most popular and most practiced sport in the world with its simplicity to play and passionate supporters. Amid the rise of the use of AI in the world for games like chess and others alike, the Robocup organization decided to address the subject of a soccer-based AI, that could one day win against a team full of actual soccer stars. The RoboCupSoccer Simulation 2D gives us the means to create such an artificial complex intelligence.

The software used for the development of the AI was Java, through Java we can send requests to the robocup server in our machine or even in another machine and thus acting as a client, send the commands we wish our player executes on the server, the server instance will then be seen as an actual game via the monitor running in parallel.

For this to happen, multiple messages are sent and received from both sides, there will be 11 clients for each team, and optionally a team coach. The messages are in different levels, they can be visual sensors of the player, gameplay status, hearing sensors, and many more as described in the manual [1] and they can also be sent between players.

Reading the manual through and trying to understand how the communication works before developing is fundamental for a good result. When the game is run, the logplayer records information of the game, so that it can be rewatched again.

A few tutorials on the subject have been created before, Carleton University [2] published a few tutorials on how to get started and get the server and client to run on a windows machine.

There are some examples of works that can be found online, for instance [reference] https://carleton.ca/nmai/krislet/ by clicking on the (Java Source) on the web page, the Krislet class can be downloaded, which acts as a client that creates a player which only tries to approach the ball and shoot it to the goal.

II. Getting Started

Setting up the environment for the development is not always obvious, there are a few steps that must be respected to have a good working server and monitor. The first thing is to know the difference between the Operational System being used, for a Linux based system the installation is usually very straight forward and does not take much configuring. For a Windows computer, there must be .exe file for the monitor and server in order to run them on windows, so the versions are not as recent as for the Linux and MacOS versions.

A. Download

Downloading the server and monitor from [3] the GitHub page, for non-windows systems is elementary.

On the Download Section there will be a link to the Monitor versions and to the Server versions. After downloading the two compressed files, place them in the same folder, for simplicity.

For our developing project, the download must be made from GitLab, by cloning the repo [4], the server and monitor files do not need to be in the same folder as our repository.

With Windows OS, the Carleton University [1] tutorial explains what to download, from the [5] Sourceforge website, click on the monitor and server folder, then other folders with version numbers will appear, then you just need to choose one of the folders and check that there is a file ending with a win.zip extension, typically the 14.0.3-win for server and 14.1.0 for monitor work just fine.

B. Installation

There is no need for installation in the case of the Windows OS.

For the other two types of Operational Systems, a few dependencies are to be installed, by following the [4] GitLab readme file, run the corresponding installation package command from the Operational System being used for each of the dependencies, g++, make, boost, flex, bison and qt5.

After all dependencies are installed and the files are extracted, run ‘./configure’ in both the directories and then run make. If it does not work on Mac, please check the qt5 installation, by running ‘brew info qt5’ you will find out where the dependency is installed, and then you just need to set up the env variables, with the LDFLAGS, CPPFLAGS, PKG_CONFIG_PATH, and writing to the ‘.zshrc’ file.

If it still does not work, try using sudo make instead. You might also need to add the command: ‘--with boost="boostinstallationpath"’

The installation can be quite tricky especially for MacOS and newer M1 ARM-based chips, so please check the readme file on our GitLab project and on the downloaded server and monitor files, which can give an insight into what is going wrong.

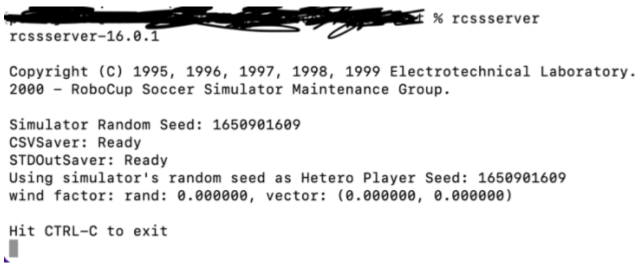
Another interesting thing to do when the installation is successful is to run ‘make install’ after running make will enable for the program to be run in the terminal in any folder, by just entering the running command

C. Running

Now the last and easiest step is to run the program, it differs for Windows OS.

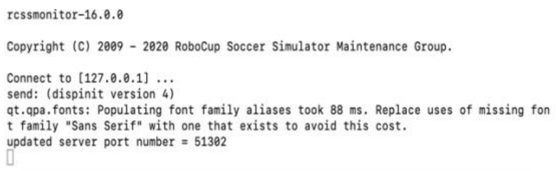
When in a Windows computer, running the server .exe file and then the monitor .exe file will start the running process. A screen will appear with the monitor, showing the game. Then by running one of our java testers in our repository, players will appear in the screen.

For the other two Operational Systems, you can either run the server and monitor separately by running rcssserver or ‘./rcssserver; if on the server folder, the following will appear:

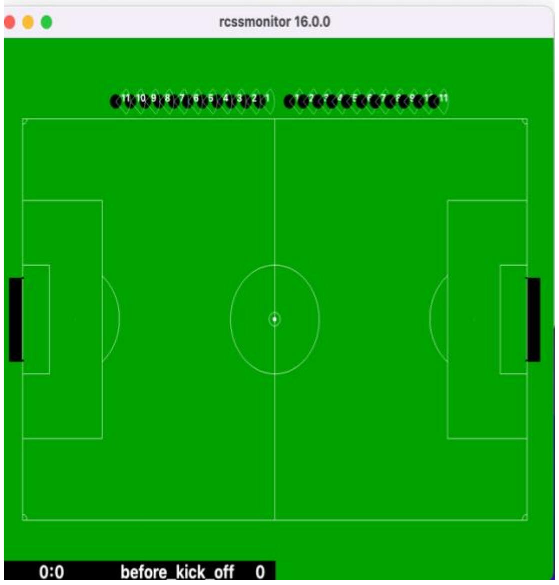


```
[redacted] % rcssserver
rcssserver-16.0.1

Copyright (C) 1995, 1996, 1997, 1998, 1999 Electrotechnical Laboratory.
2000 - RoboCup Soccer Simulator Maintenance Group.

Simulator Random Seed: 1650901609
CSVSaver: Ready
STDOutSaver: Ready
Using simulator's random seed as Hetero Player Seed: 1650901609
wind factor: rand: 0.000000, vector: (0.000000, 0.000000)

Hit CTRL-C to exit
```

Then by running the rcssmonitor or ‘./rcssmonitor’ if on the folder, the following will appear on the terminal and the screen will be instantiated.

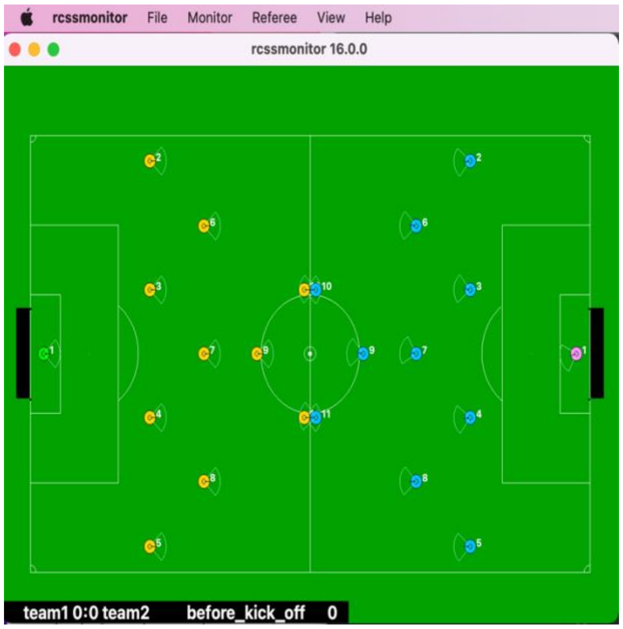


```
rcssmonitor-16.0.0

Copyright (C) 2009 - 2020 RoboCup Soccer Simulator Maintenance Group.

Connect to [127.0.0.1] ...
send: (dispinit version 4)
qt.qpa.fonts: Populating font family aliases took 88 ms. Replace uses of missing fon
t family "Sans Serif" with one that exists to avoid this cost.
updated server port number = 51302
```

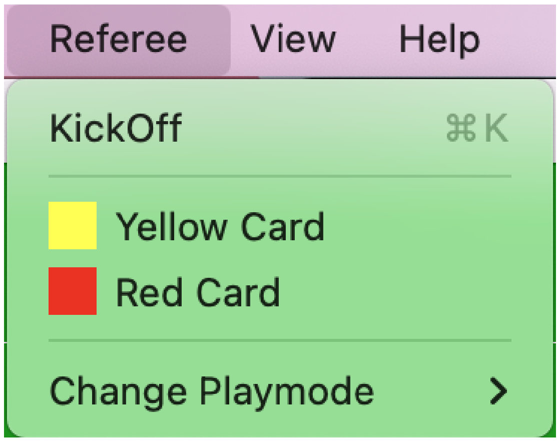


Instead of running both commands one after the other, the ‘rcsoccersim’ command can be run, which runs the server and instantiates the monitor at the same time. To stop the running, just typing control C in the terminal shuts down the process.

The next thing to do here is to run one of our testers, from our repository, navigate to RCSS2D, then run for instance the tester7.



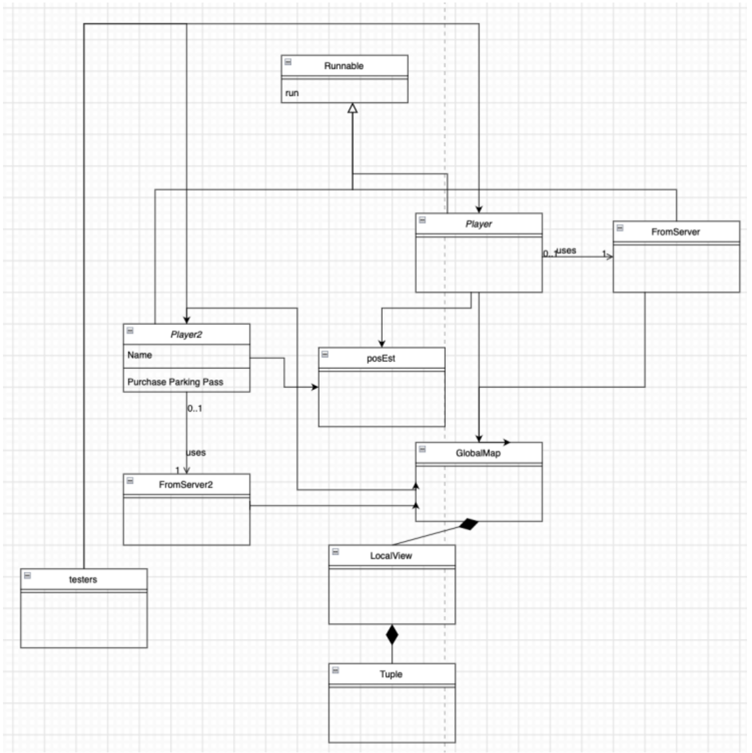
You will see that there are 11 players per side, each with their own shirt number. For the simulation to start, you will need to go on referee in the settings and then click on KickOff, then the timer will start.



Tester7 accepts input from the command line, by typing the teamname playernum anychar, the player with playernum in the corresponding team will go towards the ball and try to shoot it towards the goal. To run another tester with instances of clients, the server must be terminated and reinitiated as well as the monitor.

III. Software

The software developed for now is completely in the repository, it is all coded in Java and the classes are all placed inside the RCSS2D folder.



In this UML diagram you can see how the classes relate to each other, the functions are all well commented in the gitlab repository.

A. Classes

The main class to be used is the Player, when seen in conjunction with the FromServer they have a one to one correspondence with the client structure. Player is an instance that acts like the part of the client communicating directly with the server by sending the commands, while the FromServer is the part receiving the messages back. The Player first instantiates a client by means of the construct method, a packet is sent to the server with the initial settings for the corresponding player, including the team name and if it is a goalkeeper or not. The players get numbers assigned in ascending order, from 1 to 11, it is not possible to decide the number before instantiating them, as in the init packet sending, we can only specify if it is a goalkeeper, and which team the client is on.

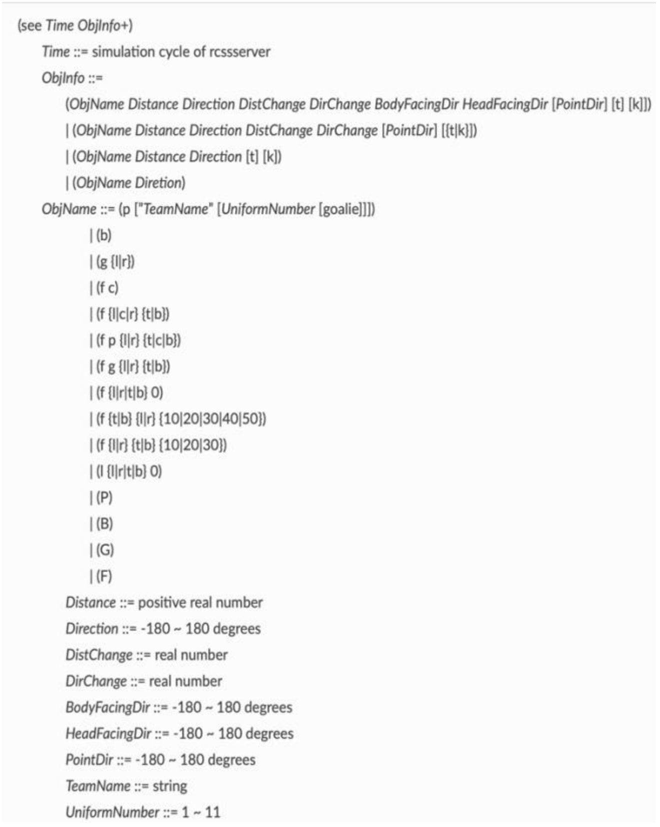
The Player class has an associated FromServer class, which hears from the server through a running thread, and updates the game information that the player sees. The information is stored in the GlobalMap variable that is associated with the Player. The FromServer class then sends the message as a LocalView to the GlobalMap through its updateMap method that calls update method on the GlobalMap. In the runnable method of the player, we will be developing our AI to send commands to the server.

GlobalMap holds information on the map around the player, like where the ball is, what the angle is between the ball and the direction the player is facing. Through LocalView instances that hold this information. The GlobalMap parses the information and stores in the LocalView as Tuples

LocalView holds information on the time the message was received from the server, the name of the objects related, and the different parameters related to them, through parsing the message received from the server.

Each Tuple correspond to an object in the LocalView time and holds the parameter information of this object.

In the image you can see how a see message is built and what can be seen by the player and stored as a tuple.



```
(see Time ObjInfo+)
    Time ::= simulation cycle of rcssserver
    ObjInfo ::=
        (ObjName Distance Direction DistChange DirChange BodyFacingDir HeadFacingDir [PointDir] [t] [k]])
        | (ObjName Distance Direction DistChange DirChange [PointDir] [{t|k}])
        | (ObjName Distance Direction [t] [k])
        | (ObjName Diretion)
    ObjName ::= (p ["TeamName" [UniformNumber [goalie]]])
            | (b)
            | (g {l|r})
            | (f c)
            | (f {l|c|r} {t|b})
            | (f p {l|r} {t|c|b})
            | (f g {l|r} {t|b})
            | (f {l|r|t|b} 0)
            | (f {t|b} {l|r} {10|20|30|40|50})
            | (f {l|r} {t|b} {10|20|30})
            | (l {l|r|t|b} 0)
            | (P)
            | (B)
            | (G)
            | (F)
        Distance ::= positive real number
        Direction ::= -180 ~ 180 degrees
        DistChange ::= real number
        DirChange ::= real number
        BodyFacingDir ::= -180 ~ 180 degrees
        HeadFacingDir ::= -180 ~ 180 degrees
        PointDir ::= -180 ~ 180 degrees
        TeamName ::= string
        UniformNumber ::= 1 ~ 11
```

In addition, a variation of the couple Player and FromServer also exists in our classes. Player2 and FromServer2 are a different approach to the running of the client, the FromServer2 does not run as a thread and is instead called by the Player2 thread every few milliseconds and gathers information from the server's pitch.

Our GlobalMap can also be visualized through a JPanel unique for each player that simulates the pitch and what the player is able to see.

B. Commands

The commands that the client sends to the server will reflect on the corresponding player, the initial commands before the kickoff are to set up the player on the pitch through the move x y command, it moves the player which is sending the command to the corresponding x and y positions, it can only be called when the PlayMode is before a kickoff or when a goal was scored. The player can also be turned before the game starts through the turn command.

Once the game starts, the move command cannot be used anymore, but the turn is still legal. To go around the pitch a client sends the dash command, that makes the player go to the direction it is facing. The kick command can be used when a player is close enough to the ball. A tackle can also be used by the player. The catch command is to be used by the goalkeeper player to catch the ball when it is in range.

In out Player class, we have a function which performs the command specified, "doAction (String command, String param)" where passing the command and params as strings will send the message from the client to the server and the result will be shown in the screen.

C. Sensory Information

When receiving messages from the server, a client receives sensory information as well as status information on the current game, like which team kicks off, or if the client's player received a card.

A Robocup agent has 3 different sensors, the aural sensor, vision sensor and body sensor. Aural sensors describe information sent from the coach referee and other players, the visual one detects the visual field information, body sensor detects the current physical status of the player.

The format received from an Aural sensor is (hear Time Sender ‘’Message’’).

Time indicates the current time that the information was received, Sender can be one of the following : self (this player), referee, online_coach_left or online_coach_right. The following table describes the messages that the referee can send to the players.

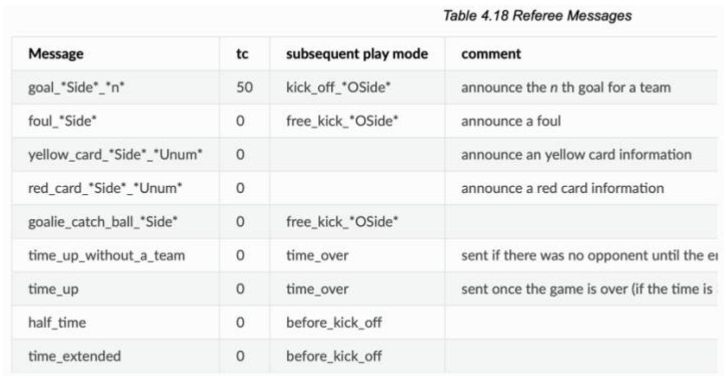


Table 4.18 Referee Messages

Message	tc	subsequent play mode	comment
goal_*Side*_*n*	50	kick_off_*OSide*	announce the *n* th goal for a team
foul_*Side*	0	free_kick_*OSide*	announce a foul
yellow_card_*Side*_*Unum*	0		announce an yellow card information
red_card_*Side*_*Unum*	0		announce a red card information
goalie_catch_ball_*Side*	0	free_kick_*OSide*	
time_up_without_a_team	0	time_over	sent if there was no opponent until the e
time_up	0	time_over	sent once the game is over (if the time is
half_time	0	before_kick_off	
time_extended	0	before_kick_off	

The vision Sensor is the most important one for our AI, it is the one that we will base our study from, and our command response.

Visual information arrives in the following format:

(see ObjName Distance Direction DistChng DirChng BodyDir HeadDir)

The Distance, Direction, DistChange and DirChng are calculated following the equations bellow:

$$
\begin{aligned}
p_{rx} &= p_{xt} - p_{xo} \\
p_{ry} &= p_{yt} - p_{yo} \\
v_{rx} &= v_{xt} - v_{xo} \\
v_{ry} &= v_{yt} - v_{yo} \\
Distance &= \sqrt{p_{rx}^2 + p_{ry}^2} \\
Direction &= \arctan\left(p_{ry}/p_{rx}\right) - a_o \\
e_{rx} &= p_{rx}/Distance \\
e_{ry} &= p_{ry}/Distance \\
DistChng &= (v_{rx} * e_{rx}) + (v_{ry} * e_{ry}) \\
DirChng &= [(-(v_{rx} * e_{ry}) + (v_{ry} * e_{rx}))/Distance] * (18 \\
BodyDir &= PlayerBodyDir - AgentBodyDir - Agent \\
HeadDir &= PlayerHeadDir - AgentBodyDir - Agent
\end{aligned}
$$

One of the most important things that we have to keep in mind when coding is how the player knows where it is located, there is no information coming from the servers that actually describes in an (x,y) axis where our players are positioned in the pitch. There are two approaches to address this situation, the first one is to keep track of all movement from the player at all moments from when it was first positioned on the pitch, such that it keeps a record and calculates following all movement from dash commands etc. where the player is located on the pitch.

The second and most efficient way is to calculate our position from the flags positioning in the pitch. In the table below you can find 55 flags positioned around the pitch, to find out the player's position we use the visual protocol that we received with the flags that are visually sensible, and calculating from the distance from them where our player is located. This is performed inside our GlobalMap calculate position function.

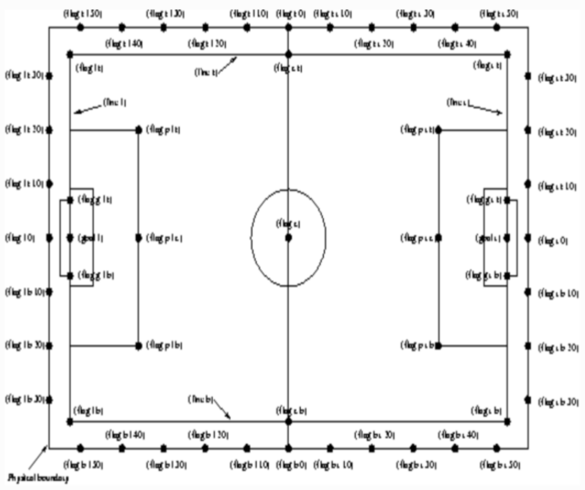


Fig. 4.2 *The flags and lines in the simulation.*

The field visualization has a couple of ranges:

If distance ≤ unum_far_length, then both uniform number and team name are visible.

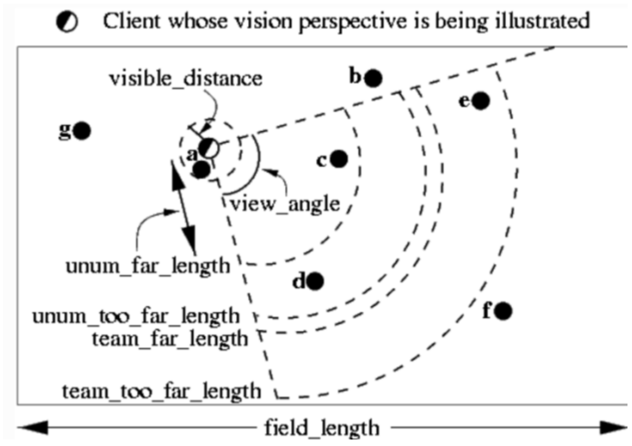
If unum_far_length <distance <unum_too_far_length, the team name is always visible, but the probability that the uniform number is visible decreases linearly from 1 to 0 as distance increases.

If dist ≥ unum_too_far_length, then the uniform number is not visible.

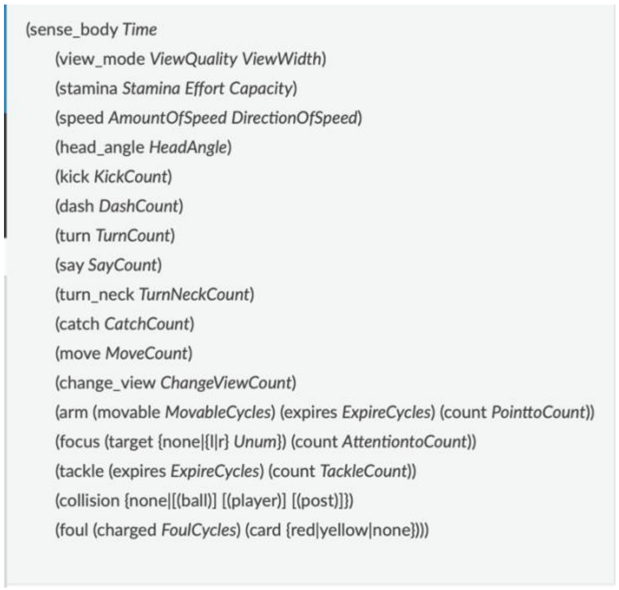
If dis <= team_far_length then the team name is visible

If team_far_length < distance < team_too_far_length then the probability that the team name is visible decreases linearly from 1 to 0 as distance increases.

If distance > team_too_far_lenght then the team name is not visible.



The body sensor informs the player about its stamina, angle, speed, command count, collision, and other body physical related sensors.



```
(sense_body Time
    (view_mode ViewQuality ViewWidth)
    (stamina Stamina Effort Capacity)
    (speed AmountOfSpeed DirectionOfSpeed)
    (head_angle HeadAngle)
    (kick KickCount)
    (dash DashCount)
    (turn TurnCount)
    (say SayCount)
    (turn_neck TurnNeckCount)
    (catch CatchCount)
    (move MoveCount)
    (change_view ChangeViewCount)
    (arm (movable MovableCycles) (expires ExpireCycles) (count PointtoCount))
    (focus (target {none|{l|r} Unum}) (count AttentiontoCount))
    (tackle (expires ExpireCycles) (count TackleCount))
    (collision {none|[(ball)] [(player)] [(post)]})
    (foul (charged FoulCycles) (card {red|yellow|none})))
```

D. Current AI

For now, the artificial intelligence in our player was not completely explored yet, as it is in its initial stages. What we were able to develop, is a way to parse the messages received from the server with the information about the player's and then have them stored in the player's GlobalMap class. This class keeps track of a series of timed see messages that were received. From the messages received our player responds according to what mode it is on.

For now, we have that the player can be on the "followball" mode, where it seeks for the ball and tries to get close to it to kick it towards the goal.

The goalkeeper mode was also in development, but was not completely tested yet, where a goalkeeper needs to catch the ball as soon as it is in the range.

The typical developing of the player AI is divided in the player's command part, and the Brain part. Where the Brain takes care of all the information that the player receives from the server, and the command part is the one that decides what to do with that information. In our case the Brain part could be seen as the GlobalMap, and the command part would be the player thread class itself.

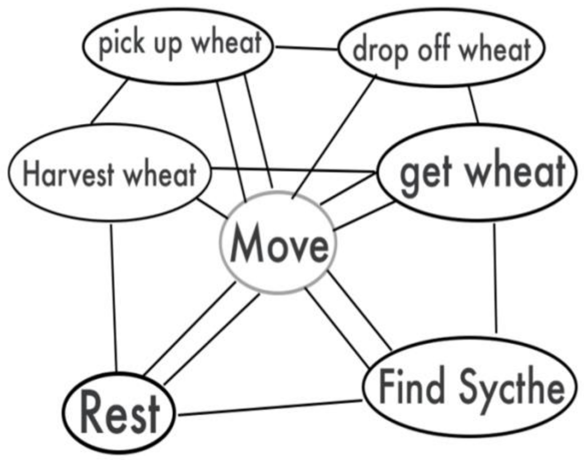
E. Future AI

The idea for our future AI would be to develop following a GOAP (Goal Oriented Action Planning) planner.

GOAP essentially means that we define different actions that the agent can do, for example run towards the ball, kick it in a direction, pass it to another player etc. It does not mean that we define the commands like dash, kick etc. but more in the direction where we use the addition of a few of these commands to define these different actions.

We then keep track of a State Vector, which keeps track of a few variables defining the environment, for instance, one variable could inform if the ball is close enough for a shot, if the agent has stamina to dash, if the player can tackle another etc.

Then from the environment's description through this state vector our agent will choose an action that has a precondition matching the current state, and that has a postcondition that is closer to the goal state as the current state. Through this system a plan of a few actions will be built with an action following the next.



This is an example of a few Finite State Machines where we can see that they are connected, each representing an action, from one of the actions we can go to a next one through its postcondition matching the next one's precondition.

F. Suggestions

Some suggestions for further developing would be to include all the three sensors in the parsing from the server, as we only handle the visual one for now.

Having a goalkeeper working would be the next step, as we can then start to play actual games, and test out with more depth how our team functions.

The objective here is to have either an AI taking care of the whole team, or of individual players.

Starting to develop without knowing the structure of the Robocup agents can be quite complicated, so having a good look at the manual, especially the Soccer Server and Client sessions is fundamental to understand what you are doing when coding.

Differentiating between different positions in the field and having different AI systems for each would be one of the final steps for finalizing our team AI.

IV. Conclusion

We started the development of an AI based Robocup player and implemented a few actions that it can do whenever we tell the player to. The complexity is not there yet, but the whole program is one step forward in the direction of an autonomous soccer team that will one day compete for McGill.

By using the theory of GOAP based AI we can try to develop an action-based system, which is really good for expansion, meaning that we can easily add new actions and change their meaning as well as the pre and post conditions which define them.

Future improvements include adding a goalkeeper behaviour, then having different behaviours for each player position and number representing it, after having those behaviours which are composed of actions, the whole GOAP AI can work quite well.

References

[1] Robo Cup Soccer Simulator. Introduction https://rcsoccersim.readthedocs.io/en/latest/introduction.html

[2] Carleton University. Tutorial 1 https://carleton.ca/nmai/wp-content/uploads/Lesson-1-Setting-Up-a-Game-of-RoboCup-Soccer-.pdf

[3] Robo Cup Soccer Simulator. Simulator https://rcsoccersim.github.io/

[4] McGill Robocup Gitlab https://gitlab.cs.mcgill.ca/jvybihal/soccer-tournament

[5] SourceForge. RobocupServer Files https://sourceforge.net/projects/sserver/files/

[6] Chaudhari V. (2017, December 12) Goal Oriented Action Planning https://medium.com/@vedantchaudhari/goal-oriented-action-planning-34035ed40d0b