Maracatronics VSS Team Description Paper for LARC 2020

Júlia D. T. Souza[1], Joaquim E. A. Araújo, Zilde S. M. Neto, Amanda S. de C. Moraes, Caio M. A. Montarroyos, Danillo do N. Santos, Flavio L. C. Oliva, Flávio M. C. da Silva, Gabriel T. S. Muniz, Guilherme S. A. e Silva, Ítalo R. A. Silva, Lucas F. Benozatti, Luís E. M. Alves, Luiz H. de S. Arruda, Marcos E. A. de Lima, Pedro S. V. Motta, Víctor H. M. Silva, Victor X. C. de Oliveira, Vinícius A. Pereira, Pedro J. L. da Silva, João P. C. Cajueiro, João M. X. N. Teixeira, José R. de O. Neto, Guilherme N. Melo

***Abstract*— This paper presents the current state of the Maracatronics team as it stands for Latin American Robotics Competitions - Very Small Size Soccer League 2020. Due to the competition being made using simulators this year, this paper contains descriptions only of controller software used as part of Armorial Project, the current project to create mobile and autonomous robots of Maracatronics Robotics Team. The description of the mechanical and electronic projects will be for the next edition of the championship, when both projects are tested for the rigors of the competition.**

I. Introduction

In 2020, since the Very Small Size Soccer category competition announced that will use a simulated environment, the team decided to start the development of the software responsible for controlling robots on the field as part of the Armorial Project, a long-term project for the development and improvement of robots autonomous.

For the software development, a base structure provided by the Warthog Robotics team from USP São Carlos, called GEARSystem, was used. Thus, the entire project was built using an architecture with distributed systems allowing the exchange of components according to the environment played.

This proved to be advantageous, because as the category does not have yet a single vision software, it is possible to change only a portion of the code responsible for receiving the data. Thus maintaining a seamless interface for the other components that will not need to be changed.

For the main control software, a layer structure was adopted with three layers that define individual attributes. However for the players and two layers responsible for defining strategies and organizing the positioning of the teams on the field.

To present the main changes in the project, this paper is organized as follows: Section II presents GEARSystem architecture used to develop the entire solution. Section III presents the hierarchy tree of Coach Coach entity. Sections IV presents algorithms used to defend out goal. Section V presents the algorithms used to path planning. Finally, Section VI presents results of preliminary tests and our open-source contributions to Robocup and LARC community.

[1]All authors are with Maracatronics Robotics Team, at Mechanical Engineering Department, Center for Technology and Geosciences, Universidade Federal de Pernambuco, State of Pernambuco, Brazil maracatronics@gmail.com

II. Control System Architecture

A. GEARSystem

We choose the GEARSystem [1] as the main framework of the project as long as it is used on the Warthog Robotics SSL team. This architecture offers some important information used along the main code at Armorial-Suassuna-VSS [2].

GEARSystem provides an architecture based on distributed systems with 3 modules and 1 server to provide communication between modules, as can be seen in Figure 1. The first module is the Sensor module which is responsible to get information from the real world through sensors (cameras) or network, filter them and delivery it to other modules. The Controller module is responsible to process the collected data and decide what each robot will execute in the field. The Actuator module is responsible to send the commands to the robots, whether simulated or real. The server or Backbone is responsible to connect the three modules and communicate them. A simple representation can be see in Figure 1.

B. Backbone (Server Module)

The Server module, or Backbone, is responsible for communicating the modules. For this purpose, it uses the omniORB library available in CORBA interfaces. There are a total of three CORBA interfaces, one for each module, and this interfaces works as the Backbone boundary, interfacing with the external elements. There are two classes in Backbone:

1) WorldMap: This class saves information from the sensor about the field geometry, players status and ball status and sets them available for the controller. To prevent conflicts from the information sent or received by the server threads, WorldMap uses Mutex to avoid conflicts during writing or reading an information.

2) CommandBus: This class saves the commands sent by the controller and sets them available to the actuator. To prevent errors from the simultaneous commands sent, it also uses a Mutex to ordinate the commands in a queue.

C. Armorial-Carrero (Sensor Module)

Our Sensor module, called Armorial-Carrero, has role of taking data from the field and treat them to make some useful stuff.

This operation uses 2 different threads to complete its objectives with one submodule running in each one. A

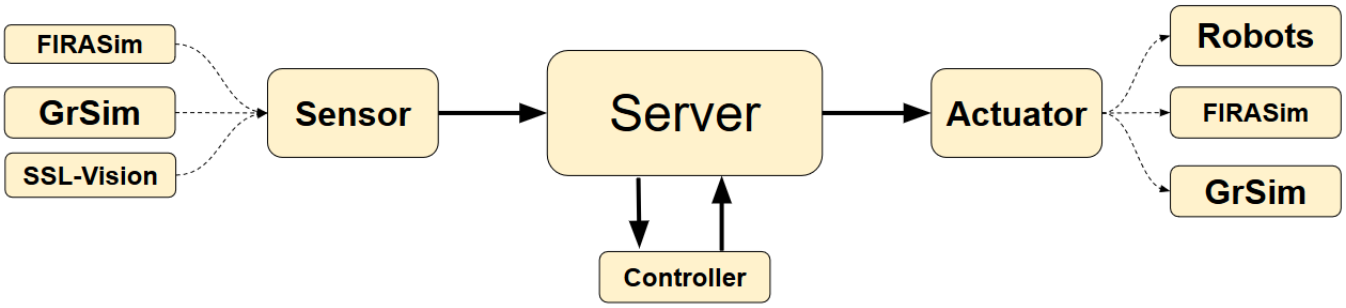

Fig. 1. GEARSystem structure.

workflow of Armorial-Carrero structure is shown in Figure 2.

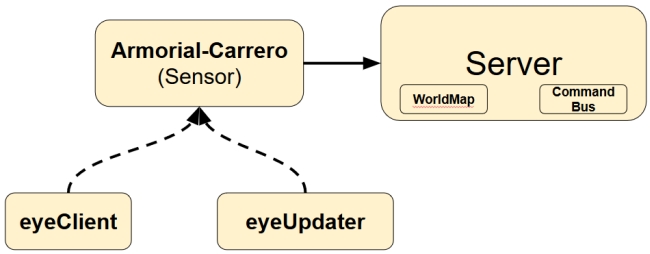

Fig. 2. Armorial-Carrero structure.

The first one is the ArmorialVisionClient which takes brute data from the field or simulator (in packages) and ordenate them for a posterior treatment..

The second one is the ArmorialVisionUpdater which takes the packages from ArmorialVisionClient and threat the informations, generating the geometry limitations, like the field, player and ball dimensions, and some interesting informations using classes available on GEARSystem, like the id of players and teams, the position of the ball and players and some relevant points at the field, like the penalty mark. Then, this information is sent to Backbone.

ArmorialVisionUpdater also uses some filters to get more precise informations:

1) Multi-Object filter: : This one guarantees the oneness of the objects.

2) Noise filter: : It prevents the apperance of invalid objects on the field

3) Loss filter: : A filter used to solve the disappearance of objects on field for any reason. It doesn't solve the problem at all, but it prevents that many objects disapppear for much time.

4) Kalman filter: This one takes some data from the vision and some data about the data the module already have and makes a prediction of the positioning of moving objects.

A full descriptions of the methods used in the submodules and the filters are available at [1].

D. Armorial-Actuator (Actuator Module)

The actuator module, called Armorial-Actuator, has the role of connecting the informations obtained from Armorial-Suassuna-VSS and send packages of information which will update the game situation on the simulator FIRASim. The connection with FIRASim occurs via the Protobuf protocol.

Armorial-Actuator has a timer information, that way connection problems can be pointed and it's a way we can verify if the packages are being sent the right way.

E. Armorial-Suassuna-VSS (Controller)

The controller module, Armorial-Suassuana-VSS, processes the current state of the game and based on that decides what actions will be taken by each robot. It receives information from Armorial-Carrero and use this values to make some standard values used along the codes. A workflow of Armorial-Suasuna-VSS structure is shown in Figure 3.

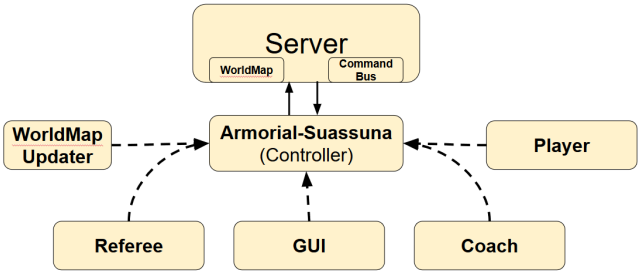

Fig. 3. Armorial-Suassuna structure.

Then, we have to make connections with the standard information, like a yellow robot has the same team id of another yellow robot, so that we can have control of (only) our players and useful information.

Armorial-Suassuna-VSS has a thread named WorldMap, which manages everything, and sub-threads:

1) Player: : This thread controls our players and its actions, like the torque applied in the wheels, the skills available to it and the information associated with it (id, team, color, etc).

2) Coach: : This thread collects the coach commands and controls the actions and formations our robots would choose in the actual game situation.

3) User Interface: : This thread controls what is seen on the User Interface which can be defined up to debug some features implemented or show the game from top as a way of game transmission.

III. Coach Architecture

The coach is the entity responsible for determining the strategy to be followed by the team. Once the strategy is defined, the team plays according to a hierarchical tree composed of an individual layer and a collective layer, each of them with multiple levels, represented by Figure 4.

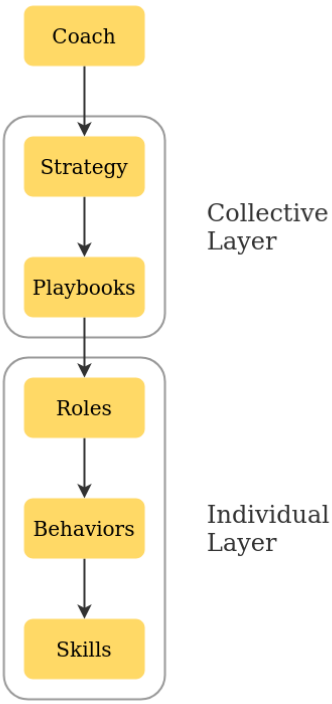

Fig. 4. Control Hierarchy Tree

A. The Individual Layer

The individual layer corresponds to the actions and set of actions that can be individually assigned to a single player. Its levels are composed by:

1) Skills: Skills are the atomic actions that can be performed by a player, such as pushing the ball, spinning and going to a specific point on the field.

2) Behaviors: A behavior can be described as a set of skills and conditions that combined form a more reactive attitude: It is able to alternate the skills according to the field elements state (ball position, ball velocity, closest enemy position etc).

3) Roles: A role is a complex set of behaviors that assigns to a player a well-defined function in the team. Some examples of roles are: attacker role, goalkeeper role etc.

B. The Collective Layer

The collective layer represents a grouping of elements from lower levels that can be assigned to a group of players. Its levels are composed by:

1) Playbooks: The playbook level is the first to enable interaction between individual elements at lower levels in the control hierarchy (following a bottom-up approach on Figure 4). A playbook is responsible for grouping roles capable of working together and for assigning them to a group of players. An offensive playbook, for example, can bring together the Attacker and Supporter roles, while a defensive one can group the Goalkeeper and Defensive Assistant roles.

2) Strategies: A strategy manages the playbooks used by the team and the number of players assigned to each one of them. Different strategies can be used depending on the state of the game and on the aggressivity module (which uses the distribution of players on the field as a parameter).

IV. Defensive Tactic

Our defensive tactic consists on reducing the score chance for the opponent team. So that, there are two main locals which we use a different heuristics: The goal area and our half side (excluding the goal area).

1) The goal area: The goalkeeper will protect that area predicting the position the ball should enter on that area. For this, it creates a line in the slope-intercept form:

$$y = a * x + b \tag{1}$$

Which coefficients are described by the equations below:

$$a = tan(\theta_{ball}) \tag{2}$$

$$b = y_{ball} - a \times x_{ball} \tag{3}$$

Where θ_{ball} is the angle of the velocity vector of the ball in the system coordinates of FIRASim. To prevent that the robot takes unwilling movements, we limit its movements at $x = 0.7$ or $x = -0.7$ (depending which is our team side) and $-0.35 < y < 0.35$. That way, the goalkeeper will remain on the goal area all the time.

2) Our half side: The middle robot will take a barrier behaviour and protect this region from plays started at the enemy half side, staying in front of the ball when an enemy has the ball possession. This way we can prevent an enemy dash advance which are the most imminent score chance nowadays. To complete this task, it creates a line the same way used by the goalkeeper, but it has a bigger range of actuation, which means it has less moviment limitations.

In both situations, the robots can spin to clear the ball and take away the danger.

V. Navigation

A. Free Angles Algorithm

The Free Angles algorithm is used to assess the presence of obstacles with respect to an observer's position and a search radius. Through the range of angles provided by this algorithm it is possible to move the robot avoiding collision with the other players on the field.

In order to develop this assessment of obstacles present in a search area, in the first step the algorithm fills in a list of obstacles that are within range with respect to a given position. Having the complete list, the obstructiona angle of the obstacle in relation to the observer is calculated for each element of the list. To perform this calculation of the angular range of obstruction, it is first necessary to calculate the angular inclination of the line segment between the position of the observer P_w and the position of the obstacle P_o, where we can calculate using the equation 4.

$$angleToObst = \arctan(\frac{Px_o - Px_w}{Py_o - Py_w}) \quad (4)$$

With this, it is necessary to calculate the Euclidean distance from the observer to the obstacle, in order to find the amplitude of the angular range of vision obstruction. To calculate the amplitude of this angular range, the obstacle geometry was approximated to a circumference of radius R_o, where the center of that circumference corresponds to its position P_o. And using the equation 6 it is possible to calculate half the amplitude of the obstruction angular range defined as *angleOffset*.

$$dist_{ow} = \sqrt{(Px_o - Px_w)^2 + (Py_o - Py_w)^2} \quad (5)$$

$$angleOffset = \arctan(\frac{R_o}{dist_{ow}}) \quad (6)$$

Finally, it is necessary to reference the initial and final angles that form the obstruction of vision of the obstacle, using the equations 7 and 8:

$$initAngleToObst = angleToObst - angleOffSet \quad (7)$$

$$finalAngleToObst = angleToObst + angleOffSet \quad (8)$$

Having all the information regarding the range of angles that are blocked by obstacles within the search area, it is now enough to include the ranges of angles that are free, that is, to fill the gaps between the ranges of obstructed angles. Therefore, at the end of the process, Free angles is able to provide a list that contains both the angle ranges that are blocked by obstacles, and the angle ranges that are free as shown in the figure 6.

B. Path Planning

To enable the robot to reach a position of interest in the field, without collisions with the other robots present, the RRT [3] navigation algorithm was implemented. This algorithm works by constructing two position trees that are generated in a random way for quick exploration that are rooted at the starting point of the robot and the desired end position. In this way, each tree must explore the environment around them and also move towards themselves.

To find the best trajectory of the robot's displacement, this method performs a sampling process with the free positions

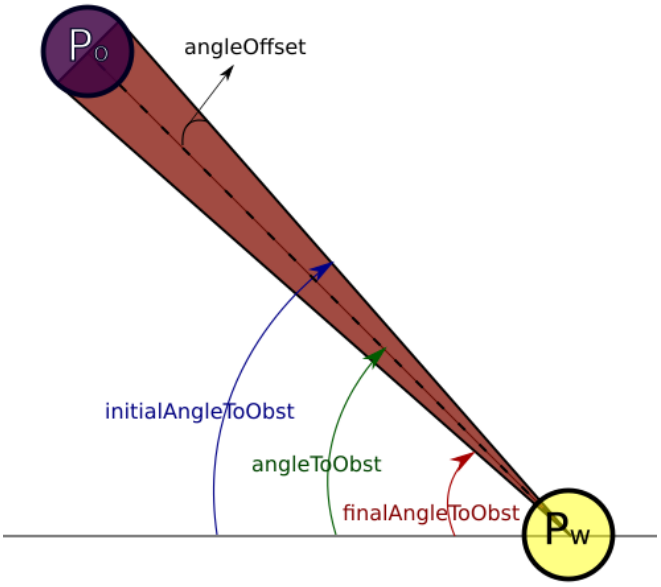

Fig. 5. Sketch of the angles that define the obstacle's angle of obstruction in relation to an observer

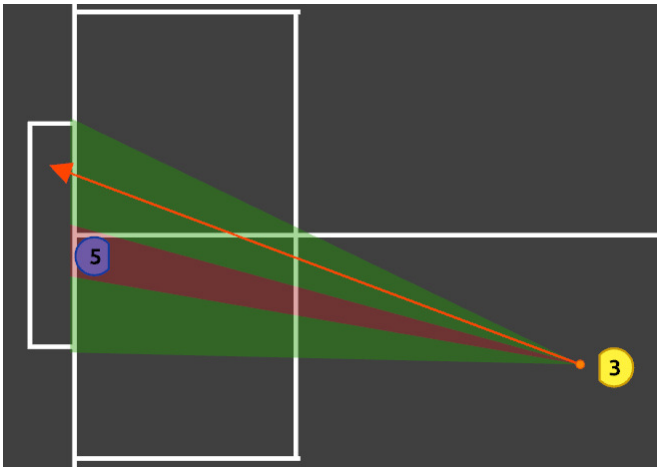

Fig. 6. Example of breaking intervals that the Free Angles algorithm performs - In red, the interval that is blocked. In green, the intervals are free intervals, and the red arrow is positioned in the middle of the largest free interval

of the field, and then traces several possible routes through these positions forming a tree graph. Having the graph calculated, the system must choose which branch of the graph has the lowest travel cost, i.e. the shortest route for the robot to reach the position of interest. A example of a path generated by RRT algorithm is shown in Figure 7.

C. Navigation Control

To allows robot to follow the trajectory defined by the path planning algorithm, it is necessary to carry out a control of the speed of each wheel. Therefore, the control of the robot's position is not only done through its linear speed, but also depends on the angular orientation of the robot at

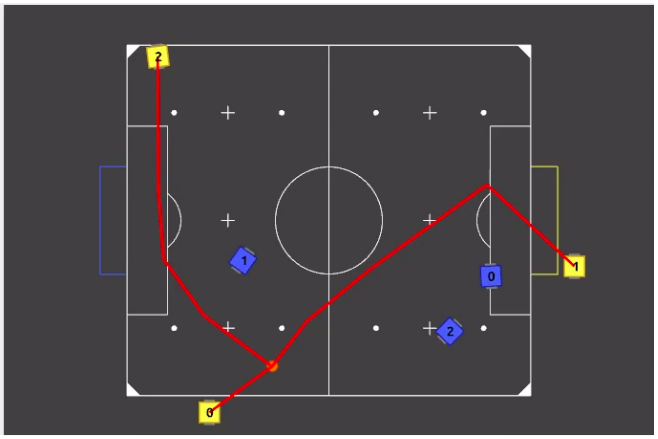

Fig. 7. Example of the path generated using the RRT algorithm

each moment of the displacement in the path.

In the figure 8 it is possible to identify the basic elements that defines the kinematic model of the robot. From the study of these elements, we can use some mathematical relationships to define the linear and angular displacement of the robot. Thus, we can define the angular speeds of each wheel using the equations 9 and 10:

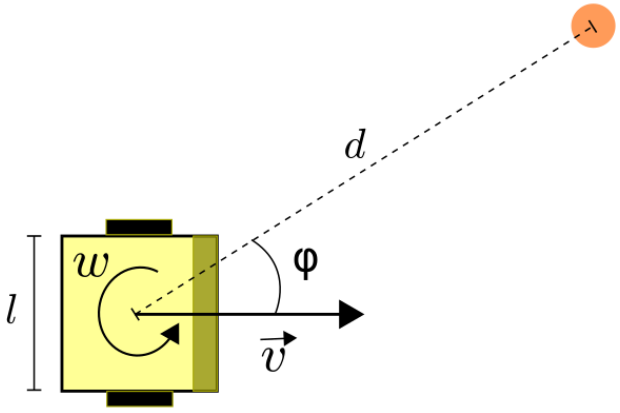

Fig. 8. Kinematic model of the robot

$$W_r = \frac{2 * v + l * w}{2r} \tag{9}$$

$$W_l = W_r - \frac{l * w}{r} \tag{10}$$

In the equations 9 and 10 w and v represent the angular and linear velocities of the robot and l and r are the distance between the wheels and wheel radius respectively.

To optimize the robot movement, we can use both the front of the robot and the back to perform the displacement. This decision on which side to use can be made with respect to the robot's angular displacement angle module (angle ϕ in the figure 8), since this is greater than 90°it would be more advantageous to use the back of the robot to perform the displacement.

VI. Conclusion

Preliminary tests using the FIRASim simulator, the code showed good results managing to perform from simple tasks such as avoiding obstacles and moving towards the ball to more complex tasks such as playing with the complete team against another complete team.

Looking at the 5x5 challenge, the structure presents features that will allow the software to scale without major problems as well as the creation of new roles for the players, allowing for a more dynamic game and for the players to have acting specialties just like in human football.

All the project's source code is available on the team's Github with documentation for the community.

References

[1] R. G. Lang, I. N. da Silva, and R. A. Romero, "Development of distributed control architecture for multi-robot systems," in *2014 Joint Conference on Robotics: SBR-LARS Robotics Symposium and Robocontrol*. IEEE, 2014, pp. 163–168.

[2] Maracatronics, "Armorial-suassuna vsss repository," https://github.com/MaracatronicsRobotics/Armorial-VSS, 2020.

[3] J. J. Kuffner and S. M. LaValle, "Rrt-connect: An efficient approach to single-query path planning," in *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No. 00CH37065)*, vol. 2. IEEE, 2000, pp. 995–1001.