Universidade de Aveiro Departamento de Electrónica, Telecomunicações e Informática,
2014

Ricardo Jorge Maurício Dias

Arquitetura do Agente da equipa de Futebol Robótico CAMBADA

Agent Architecture of the CAMBADA Robotics Soccer Team

Universidade de Aveiro Departamento de
Electrónica, Telecomunicações e Informática,
2014

Ricardo Jorge Maurício Dias

Arquitetura do Agente da equipa de Futebol Robótico CAMBADA

Agent Architecture of the CAMBADA Robotics Soccer Team

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia Electrónica e Telecomunicações, realizada sob a orientação científica de Prof. Doutor António Neves e Prof. Doutor Nuno Lau, professores do Departamento de Electrónica, Telecomunicações e Informática da Universidade de Aveiro

o júri

presidente

Luís Seabra Lopes
Professor Associado da Universidade de Aveiro

vogais

Eduardo Alexandre Pereira da Silva
Professor Adjunto do Instituto Superior de Engenharia do Porto

António José Ribeiro Neves
Professor Auxiliar da Universidade de Aveiro (orientador)

Agradecimentos

Pelo tempo despendido na orientação do trabalho aqui apresentado e por estarem sempre disponíveis para qualquer dúvida, agradeço aos professores António Neves e Nuno Lau. Ao todos membros da equipa CAMBADA (e IRIS no geral), um grandioso obrigado por tudo o que aprendi ao longo destes cinco anos, pelas ideias e pela confiança depositada em mim, suporte e ambiente propício ao trabalho, mas sobretudo pelo companheirismo e espírito de equipa.

Quero agradecer o apoio incondicional da minha família, em especial aos meus pais, ao longo da minha caminhada nestes 5 anos. Também aos meus amigos mais próximos e colegas de curso (particularmente do Grupo 5 - Diana, Renato e Riscado), obrigado pela vossa força e ajuda ao longo desta jornada conjunta!

Por último, um obrigado à Cátia pela paciência para me aturar nos momentos mais adversos.

Resumo

O agente de software é o processo onde reside toda a componente de Inteligência Artificial, responsável por tomar as decisões de alto nível. CAMBADA é a equipa de futebol robótico do grupo de investigação IRIS, da unidade de investigação IEETA, da Universidade de Aveiro que participa na Liga dos Robôs Médios do RoboCup.

A robótica é uma área multidisciplinar emergente que junta ciências da computação, eletrónica e mecânica e nesta tese está incluída uma explicação geral sobre a arquitetura dos robôs CAMBADA, desde o *hardware* ao *software*, sobre os quais foi desenvolvido todo o trabalho apresentado. No contexto de competição, a capacidade de raciocínio é o que define o sucesso ou o insucesso das equipas. Dado o dinamismo atual dos jogos, torna-se vital tomar as decisões corretas, no momento certo e em equipa. Com esta tese pretende melhorar-se a estrutura do agente, desde a organização do código à própria arquitetura de software. Um novo modelo de gestão de comportamentos foi desenvolvido e adotado para as competições.

A constante evolução da Liga de Robôs Médios leva as equipas a terem de se adaptar a novas regras todos os anos. Neste contexto, alguns comportamentos foram desenvolvidos de raíz e outros foram melhorados na nova arquitetura. No entanto, para a criação, teste e validação destes comportamentos foi necessária a criação de aplicações de suporte ao desenvolvimento, calibração e de depuração.

A nova arquitetura permitiu um desenvolvimento mais rápido e robusto de comportamentos, e os avanços nos comportamentos levaram a uma melhoria considerável no desempenho global da equipa em termos competitivos.

Abstract

The software agent is the process where all the Artificial Intelligence resides and is responsible for taking high-level decisions. CAMBADA is the robotics soccer team of the IRIS research group, from IEETA, University of Aveiro, that participates in the Middle-Size League of RoboCup.

Robotics is an emerging multidisciplinary area that joins computer science, electronics and mechanics and this thesis includes an overview on the general architecture of the CAMBADA robots, from *hardware* to *software*, over which all the presented work has been developed. In the competitions context, the reasoning capabilities define the success or the failure of a team. Given the high dynamism of the games, it becomes vital to take the correct decisions, at the right time and in a collaborative way. This thesis intends to improve the structure of the agent, from the code organization to the actual software architecture. A new behavior management model was developed and adopted for the competitions.

The constant evolution of the Middle-Size League pushes teams to adapt to new rules each new year. In this context, some novel behaviors were developed and others have been refined in the new architecture. Moreover, for the creation, test and validation of these behaviors, the creation of a series of applications was needed for development, calibration and debugging.

The new agent architecture provided a faster and more robust behavior development, and the improvements made on behaviours led to a better global performance of the team in the competitions.

Contents

Contents **i**

List of Figures **v**

List of Tables **ix**

1 Introduction **1**

1.1 RoboCup . . . 1
1.1.1 RoboCup Logistics . . . 2
1.1.2 RoboCup@Work . . . 3
1.1.3 RoboCup Rescue . . . 4
1.1.4 RoboCup Junior . . . 5
1.1.5 RoboCup@Home . . . 6
1.1.6 RoboCup Soccer . . . 7
1.2 RoboCup Soccer - Middle-Size League . . . 8
1.3 CAMBADA Team . . . 10
1.3.1 CAMBADA General Architecture . . . 12
1.3.2 Hardware Description . . . 15
Vision System . . . 15
Main Processor Unit . . . 17
Low-Level Layer . . . 17
Ball Handling Mechanism . . . 18
Ball Kicking Mechanism . . . 19
Motion Hardware . . . 20
1.4 Coordinate Systems . . . 21
1.5 Motivation and Main Contributions . . . 21
1.6 Thesis Structure . . . 22

2 Robotic Agent Architectures **23**
2.1 Software Agent Architectures 23
2.2 Multi-Agent Systems 24
2.2.1 MAS Example Applications 25
2.3 Coordination and Decision in MSL 26
2.4 CAMBADA Agent Architecture in 2013 27

3 CAMBADA Agent Architecture **31**
3.1 DriveVector 32
3.1.1 Interface 33
3.2 Controllers 33
3.2.1 CMove 34
3.2.2 CArc 34
3.2.3 CRotateAroundTheBall 35
3.2.4 CRotate 35
3.3 Behaviors 35
3.3.1 Interface 36
3.3.2 List of Current Behaviors 36
Generic Behaviors 37
Ball Handling Behaviors 37
Behaviors Without Ball 37
Striker Behaviors 37
Midfielder Behaviors 38
Goal-Keeper Behaviors 38
Replacer Behaviors 38
3.4 Arbitrator 39
3.4.1 Priority Arbitrator 39
3.4.2 Finish Plan First Arbitrator 41
3.4.3 Combining Both Arbitrator Types 41
3.4.4 Inheritance 43
3.5 Roles 43
3.5.1 The Role Striker Example 43
3.6 HeightMap Class 44
3.6.1 Interface 44
3.6.2 Cost Maps 46
Ball Field Of View Map 46

Goal Field Of View Map 47
Obstacles Map 47
Dribble Map 48
Kick2Goal Map 48
ReceiveBallFP Map 48
3.7 Conclusion 50

4 New and Improved Behaviors 53
4.1 Ball Reception Behavior - BReceiveBall 53
4.1.1 Results 55
4.2 Dribble - BBallBodyProtect 55
4.2.1 Implementation 56
4.2.2 Results 58
4.3 Free-Play Ball Passes 59
4.3.1 Receiver 60
4.3.2 Passer 61
4.3.3 Forward Passes 61
4.3.4 Results 62
4.4 Kick to Goal - BKickToTheirGoal 62
4.4.1 Compensating the Linear Velocity 63
4.4.2 Compensating the Angular Velocity 65
4.4.3 Results 66
4.5 Contouring Obstacles - BStrikerContourObstacle 66
4.5.1 Implementation 67
4.5.2 Results 69

5 Debug and Development Tools 71
5.1 Basestation 3D Field View 71
5.1.1 3D Models 73
5.1.2 Colors 73
5.1.3 Set-pieces 74
5.1.4 Widget Integration 74
5.1.5 Interaction 75
5.1.6 Height Map Visualization 75
5.1.7 Results and Future Work 76
5.2 Augmented Reality Visualization 77

5.2.1 Marker Detection 79
5.2.2 Interaction for Development Support 80
5.2.3 Results 81
5.2.4 Future Work 82

6 Conclusion and Future Work **85**

6.1 Conclusion 85
6.2 Future Work 88

Bibliography **89**

List of Figures

1.1 The RoboCup Logistics challenge. 3
1.2 The platform used in RoboCup@Work - Kuka YouBot. 3
1.3 Rescue robot from Darmstadt University of Technology. 4
1.4 A RoboCup Junior Soccer match. 5
1.5 AMIGO robot from Eindhoven University of Technology. 6
1.6 Middle-Size League. 8
1.7 Small-Size League. 8
1.8 Standard Platform League. 8
1.9 Simulation 3D. 8
1.10 Humanoid Kid-Size. 8
1.11 Humanoid Adult-Size. 8
1.12 MSL Field Dimensions 9
1.13 A typical MSL match setup 10
1.14 Overview of a CAMBADA robot architecture. 12
1.15 A RtDB setup example with three agents. 13
1.16 Processes running on each robot and on the basestation. 14
1.17 A CAMBADA Robot 15
1.18 CAMBADA vision system. 16
1.19 CAMBADA Main Processor Unit. 17
1.20 Low-level layer architecture. 18
1.21 The ball handling mechanism. 18
1.22 The solenoid of the kicker system. 19
1.23 The kicker shaft hitting the lever. 19
1.24 Kicker servo-motor in lob-shot mode. 19
1.25 Kicker servo-motor in pass mode. 19
1.26 One of the wheels partially assembled. 20
1.27 Power Transmission System. 20

1.28 The relative coordinate system used by CAMBADA. 21
1.29 The CAMBADA team robots 21

2.1 FACET screenshot displaying traffic routes and air flow statistics. 26
2.2 An example of a rescue situation in a simulated city. 26
2.3 Tech United task utility field 27

3.1 Comparison between the old and new software agent architectures. 31
3.2 Example of the Priority Arbitrator. 40
3.3 Behaviors added for the RoleStriker. 43
3.4 The example situation for illustration of the cost maps. 47
3.5 The Ball FOV map. 49
3.6 The Goal FOV map. 49
3.7 The Obstacles cost map. 49
3.8 The Dribble cost map. 49
3.9 The Kick2Goal cost map. 49
3.10 The ReceiveBallFP map. 49

4.1 BReceiveBall graphical representation. 55
4.2 Representation of the angular cost map for a single obstacle. 56
4.3 Representation of the angular cost map for two near obstacles. 57
4.4 Ball Possession in RoboCup 2013. 59
4.5 Ball Possession in Robotica 2014. 59
4.6 Forward Pass Algorithm. 62
4.7 Representation of the kicking alignment algorithm. 63
4.8 Influence of robot's x-velocity in kicking direction. 64
4.9 Influence of robot's angular velocity in kicking direction. 65
4.10 Allowed error vs Angular Velocity. 65
4.11 BStrikerContourObstacle contour algorithm. 68
4.12 BStrikerContourObstacle protective behavior. 68
4.13 BStrikerContourObstacle protective behavior without knowing ball position. 69

5.1 The 2D Basestation 72
5.2 3D Models created for the new Basestation Field Widget. 73
5.3 Robot 3 has the ball engaged. 74
5.4 A Free-Kick set-piece example. 74
5.5 The 3D Field Widget integrated into the Basestation. 75

5.6 Colored Height Map in flat mode. 76
5.7 The two layers in VTK for the AR application. 77
5.8 World axis centered in the pattern. 78
5.9 Coordinate systems used in ARToolkit. 79
5.10 Intersection of the optical center line with the x-y plane as the target. . . . 80
5.11 The setup used to test the application. 81
5.12 Final result of the AR application. 82

List of Tables

1.1 Summary of team results in Tournaments. 11
1.2 Summary of team results in RoboCup Challenges 12

3.1 Roles source lines of code comparison between architectures. 50

4.1 Measurements on the output angle for different robot's x-velocities. 65
4.2 Kick accuracy during IranOpen2014. 66
4.3 Kick accuracy during Robotica2014 67

Chapter 1

Introduction

Multi-Agent Systems is a research field in computer science that has been gaining much interest since the mid-1990s. A Multi-Agent System consists of more than one agents, each one being a computer system capable of executing tasks autonomously, but working together to achieve a common goal. Therefore, these agents have to **cooperate**, **coordinate** and **negotiate** with each other.

In this Chapter, an introduction to RoboCup is made, with focus on the Middle-Size League, where the CAMBADA team participates. The General Architecture of the robots is described, leading us to the motivation for this thesis: restructure the CAMBADA agent code, streamlining the whole process of developing, testing, debugging and refining agent behaviors. In the context of the RoboCup competitions, some new behaviors were also developed, due to new rules requirements for the league. In order to efficiently develop, test and validate the new behaviors, there was also some work done on debugging and calibration tools.

1.1 RoboCup

RoboCup [1] is a huge commitment from scientific and research groups all over the world on developing better and smarter robots that one day will be able to help us (humans) in our daily lives. Robocup is also a series of competitions that are held in a different country each year since 1997. The main objective is not only to create challenging environments that give an extra motivation to the teams to keep participating every year in the competitions, showing new abilities and sometimes unexpected improvements, but also a discussion forum for researchers in the robotics domain, the Symposium, in which teams have the chance to share ideas, knowledge and implementations, so the technology can improve faster.

By organizing an annual event, RoboCup creates opportunities for researchers to test their work and exchange technical and scientific information. At the same time, it entertains and educates the public, taking people to be more aware of the state of the art in robotics and its current problems. Moreover, with the ambition of keeping a high level of interest and competitiveness, the RoboCup Federation has even created a challenging milestone:

> *"By the middle of the 21st century, a team of fully autonomous humanoid robot soccer players shall win a soccer game, complying with the official rules of FIFA, against the winner of the most recent World Cup."*

We are clearly very far from this goal, but a significant progress has been made so far. The competition is divided into several leagues, each one tackling a specific problem, whether at the software level, or on both hardware and software, in a multi-agent or single-agent environments, cooperative and/or competitive:

- RoboCup Logistics - robots for logistics problems
- RoboCup@Work - industrial robots
- RoboCup Rescue - robots for disaster scenarios
- RoboCup Junior - robots for education
- RoboCup@Home - service robots
- RoboCup Soccer - robots that play soccer

Each league is detailed in the following sections.

1.1.1 RoboCup Logistics

In this league, a team of three Robotino[1] robots has to solve logistic challenges of an almost unknown production system. Participants are allowed to add any sensors they need and face no limitations regarding their approach to program the robots.

> *"This interdisciplinary challenge in the fields of Mechatronics, Computer Science and Logistics has to be answered with a flexible yet precise autonomous solution to deal with out-of-order machines, express goods, changing delivery gates and a random machine distribution."* [2]

[1] `http://www.festo-didactic.com/int-en/services/robotino/` - accessed at 19-Jun-2014

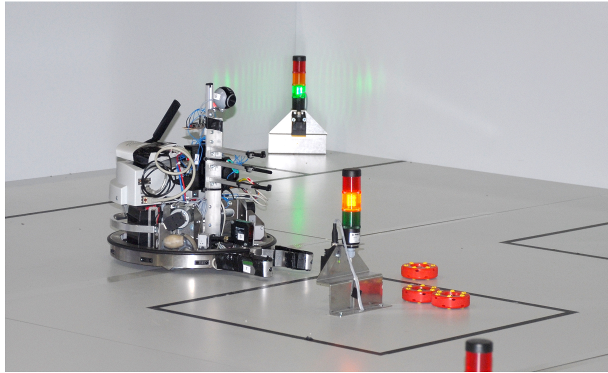


Figure 1.1: The RoboCup Logistics challenge.

1.1.2 RoboCup@Work

RoboCup@Work targets specific challenges that have not been pursued by other leagues, namely cooperative mobile manipulation, multi-agent scheduling and multi-criteria optimization. The solutions are aimed at industrial applications, where robots cooperate with human workers for complex tasks ranging from manufacturing, automation, and parts handling up to general logistics. Figure 1.2 shows the type of robots used in this league.

> *"RoboCup@Work aims to foster research and development that enables use of innovative mobile robots equipped with advanced manipulators for current and future industrial applications, where robots cooperate with human workers for complex tasks ranging from manufacturing, automation, and parts handling up to general logistics."* [3]

Figure 1.2: The platform used in RoboCup@Work - Kuka YouBot.

1.1.3 RoboCup Rescue

In the RoboCup Rescue, teams are faced with disaster scenarios and related challenges: mobility, sensory perception, mapping and planning.

> *"Disaster rescue is one of the most serious social issue which involves very large numbers of heterogeneous agents in the hostile environment."* [4]

The Rescue league includes two sub-leagues:

- Rescue Robots (Figure 1.3)
- Rescue Simulation:
  - Agent Simulation
  - Virtual Robots

In the Rescue Robots league, teams are required to search and rescue victims in unstructured environments.

In Rescue Simulation, a virtual environment is provided by a simulator where multiple intelligent software agents have to cooperate to in order to achieve the common objectives in a disaster response scenario.

Figure 1.3: Rescue robot from Darmstadt University of Technology.

1.1.4 RoboCup Junior

RoboCup Junior is an educational league for young students to participate and take their first contact with robotics. Many professors take this opportunity to teach electronics and programming in school projects to their students.

> *"RoboCupJunior is designed to introduce RoboCup to primary and secondary school children, as well as undergraduates who do not have the resources to get involved in the senior leagues yet."* [5]

There are currently three leagues in RoboCup Junior:

- Soccer Challenge (Figure 1.4)
- Dance Challenge
- Rescue Challenge

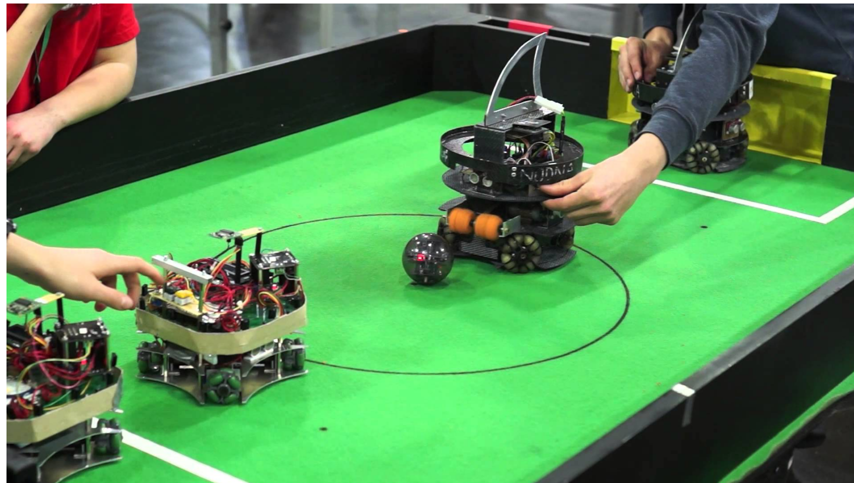


Figure 1.4: A RoboCup Junior Soccer match.

In the Soccer Challenge, students are required to design and program two robots to compete against an opposing pair of robots by kicking an infra-red transmitting ball to the opponent's goal. In RoboCup Junior Dance Challenge, participants have to program their robots to move to music or a soundtrack. Finally, the Rescue Challenge mirrors the real life use of robots in disaster situations, since they are required to identify victims in different scenarios from line-following on a flat surface to mazes on uneven terrain.

1.1.5 RoboCup@Home

This league started in 2006 and is aimed at the development and deployment of service robots for domestic applications. These assistive robot technologies are put to the test in a realistic environments created specifically for specific challenges that arise in this domain.

> *"The RoboCup@Home aims to offer a combination of interdisciplinary community building, scientific exchange and competition, which iteratively defines benchmarks and performance metrics on which service robots can be evaluated and compared in a realistic, dynamic and non-standardized domestic environment."* [6]

Figure 1.5 shows Autonomous Mate for IntelliGent Operations (AMIGO) [7], an example of a typical service robot developed mainly to help elderly people. This 1.5 meters high robot has two arms that enable it to execute several tasks and has a set of three omnidirectional wheels to move. The vision system is composed of a Microsoft Kinect camera, which enables it to see in 3D.

Figure 1.5: AMIGO robot from Eindhoven University of Technology.

Currently, the scenario consists in a living room and a kitchen, but there are plans to evolve to other social areas of the daily-life, such as shops, streets, gardens and parks.

1.1.6 RoboCup Soccer

An interesting fact is that RoboCup is a contraction of the competition's full name, "Robot Soccer World Cup". Soccer was the obvious choice: it moves masses, it engages people and it is easy for anyone visiting the venue to understand the purpose of these robots.

> *"RoboCup (Soccer) is an attempt to foster AI and intelligent robotics research by providing a standard problem where wide range of technologies can be integrated and examined. In order for a robot team to actually perform a soccer game, various technologies must be incorporated including: design principles of autonomous agents, multiagent collaboration, strategy acquisition, realtime reasoning, robotics, and sensor-fusion."* [1]

This league includes a series of sub-leagues:

- Middle-Size (Figure 1.6)
- Small-Size (Figure 1.7)
- Standard Platform (Figure 1.8)
- Simulation:
  - Simulation 2D
  - Simulation 3D (Figure 1.9)
- Humanoid:
  - Teen-Size
  - Kid-Size (Figure 1.10)
  - Adult-Size

Just like in human soccer, the players must work as a team in order to win and defend their own goal at the same time, while adapting to new situations and different opponents. This makes soccer a rich domain for fundamental exploration and development of multi-agent solutions related to perception, sensor fusion, world state representation, high-level decision and coordination and communications.

Figure 1.6: Middle-Size League.

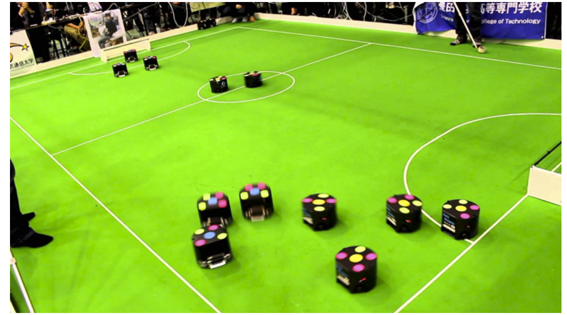


Figure 1.7: Small-Size League.

Figure 1.8: Standard Platform League.

Figure 1.9: Simulation 3D.

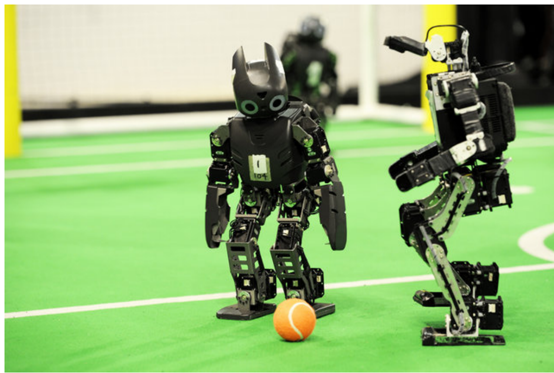


Figure 1.10: Humanoid Kid-Size.

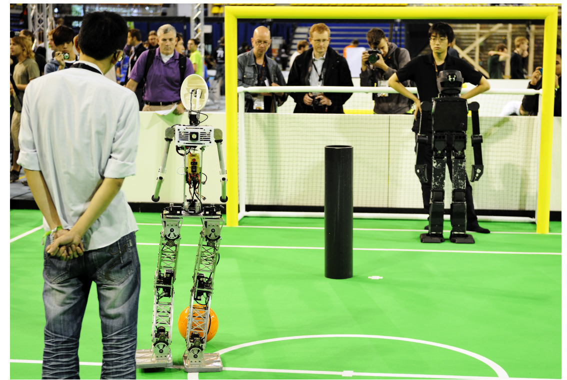


Figure 1.11: Humanoid Adult-Size.

1.2 RoboCup Soccer - Middle-Size League

In the context of RoboCup Soccer, the Middle-Size League is one of the most challenging and also a great testbed for multi-agent systems research, due to its rich and dynamic environment. In this league, the robots have no standard format, but must comply with some dimension and weight limits: they must fit in a 50cm x 50cm x 80cm box and are not

allowed to weight more than 40Kg. Each team can have up to five robots (including the goal-keeper) in a 18m x 12m field (Figure 1.12), playing soccer completely autonomously for two halves of 15 minutes. Teams are identified by a color (either cyan or magenta) and the robots wear body markers with that color and the number of the robot.

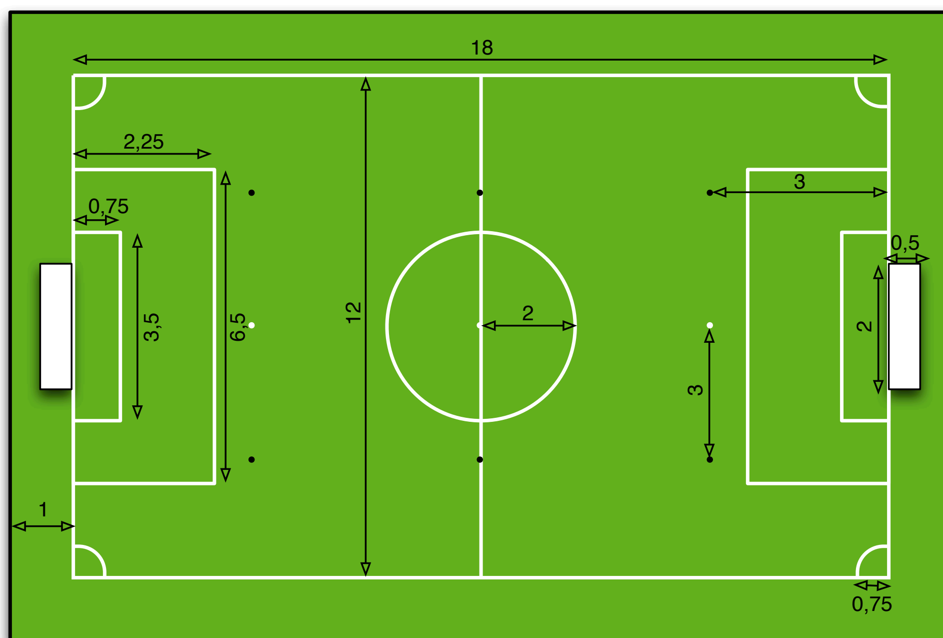


Figure 1.12: Dimensions of the MSL field, in meters, **not** in proportion.

Because human intervention is forbidden, excluding the referees and for removing or putting robots on the field, each robot needs sensors to perceive the world around it, a computer for reasoning and actuators to interact with that world.

These robots have to play as a team, so they require a mean of communication for coordination purposes. Communication among robots is established by the rules to be done via WiFi (802.11a or 802.11b), with limited bandwidth to be spent by each team. The robot's network device is usually part of the computer, since most of the teams use laptops on-board.

Teams also receive commands from a so-called "referee-box", which is an application running on a separate computer, responsible for making the bridge between the human referee and the teams.

Each team has a **basestation**, a computer application for visualization of the state of the world perceived by the robots, running on a computer connected to the referee-box and also to the team's robots. In these basestation computers, teams are able to get realtime

feedback from the robots and usually run a process that can be compared to a coach in human soccer.

Referee's decisions (start, stop, foul, goal, etc.) are sent to the teams basestations, which then communicate the state of the game to the robots. Figure 1.13 shows a typical setup during an MSL match in the competitions.

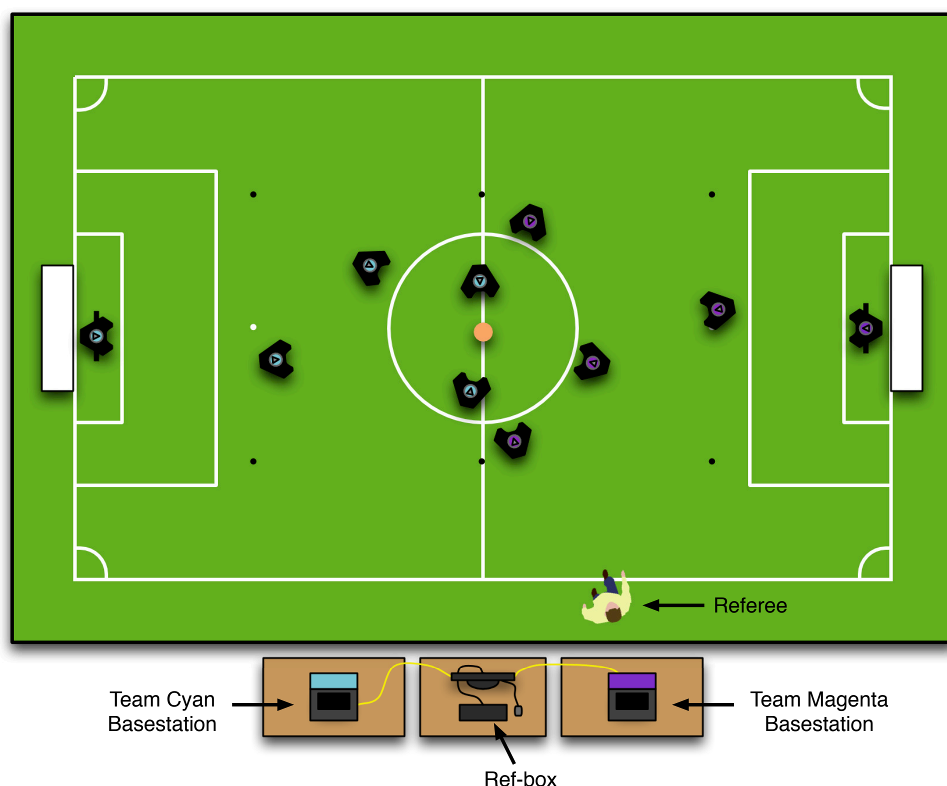


Figure 1.13: A typical MSL match setup. Two teams of five robots on the field. Below, the "referee-box" standing between each team's basestation.

1.3 CAMBADA Team

Cooperative Autonomous Mobile roBots with Advanced Distributed Architecture (CAMBADA) [8] is the RoboCup Middle-Size League robotic soccer team from the University of Aveiro. The project involves people working on several different areas: mechanical structure of the robot, hardware architecture and controllers, software on image analysis and processing, sensor fusion and decision and cooperation architecture.

The team development started in 2003 and a steady progress was observed since then, with the participation in several competitions, both national and international, including

RoboCup World Championships and the annual Portuguese Open Robotics Festival. Table 1.1 shows a summary of participations and the respective results on the tournaments.

Year	Competition	Dates	Location	Result
2004	**Robotica**	April 22 - April 25	Porto, Portugal	5th Place
	RoboCup	June 27 - July 5	Lisbon, Portugal	1st Round
2005	**Robotica**	April 29 - May 1	Coimbra, Portugal	4th Place
2006	**DutchOpen**	April 7 - April 9	Eindhoven, Netherlands	7th Place
	Robotica	April 28 - May 1	Guimarães, Portugal	3rd Place
	RoboCup	June 14 - June 20	Bremen, Germany	1st Round
2007	**Robotica**	April 27 - April 30	Paderne, Portugal	1st Place
	RoboCup	June 30 - July 10	Atlanta, USA	5th Place
2008	**Robotica**	April 2 - April 6	Aveiro, Portugal	1st Place
	RoboCup	July 14 - July 20	Suzhou, China	1st Place
2009	**Robotica**	May 6 - May 10	Castelo Branco, Portugal	1st Place
	RoboCup	June 29 - July 5	Graz, Austria	3rd Place
2010	**Robotica**	March 25 - March 28	Batalha, Portugal	1st Place
	GermanOpen	April 15 - April 18	Magdeburg, Germany	2nd Place
	RoboCup	June 19 - June 25	Singapore	3rd Place
2011	**Robotica**	April 6 - April 10	Lisbon, Portugal	1st Place
	RoboCup	July 5 - July 11	Istanbul, Turkey	3rd Place
2012	**Robotica**	April 11 - April 15	Guimarães, Portugal	1st Place
	DutchOpen	April 25 - April 29	Eindhoven, Netherlands	3nd Place
	RoboCup	June 18 - June 24	Mexico City, Mexico	4th Place
2013	**Robotica**	April 25 - April 28	Lisbon, Portugal	2nd Place
	RoboCup	June 27 - June 30	Eindhoven, Netherlands	3rd Place
2014	**IranOpen**	April 9 to April 11	Tehran, Iran	2nd Place
	Robotica	May 14 to May 18	Espinho, Portugal	2nd Place
	RoboCup	July 19 - July 24	João Pessoa, Brazil	N.A.

Table 1.1: Summary of team results in Tournaments.

Apart from the soccer tournament, there are two additional challenges in RoboCup with special requirements: the **Scientific** and **Technical Challenges**. The achievements of the team in these Challenges are summarized in Table 1.2.

The **Scientific Challenge** requires teams to do a small pitch about relevant developed

Competition	Challenge	Result
RoboCup 2011	**Scientific Challenge** - ”Formation in CAMBADA”	1st Place
RoboCup 2012	**Scientific Challenge** - ”Flexible Setplays in MSL”	1st Place
	Technical Challenge	1st place
RoboCup 2013	**Scientific Challenge** - ”Dynamic Strategy”	3rd Place
	Technical Challenge	1st place

Table 1.2: Summary of team results in RoboCup Challenges

work in the context of MSL for other teams to evaluate several criteria: presentation, novelty, interest for the league, scientific/technical complexity, importance of experimental results and relevance of the published results presented as a support for this challenge.

In the end, all weighted averages given by other teams are summed and the score ranking determines the places in this challenge.

On the other hand, the **Technical Challenge** has been an opportunity for teams to test, improve and show their coordination abilities. With some variations, this challenge has been focused on a sequence of tasks that require multiple robots. In 2014, the rules have changed to being able to play in an uneven terrain, with the objective of letting teams do some experiments in different terrains and seek the respective difficulties.

1.3.1 CAMBADA General Architecture

An overview of the general architecture of a CAMBADA robot has been thoroughly described in [9, 10, 11] and the main blocks are illustrated in Figure 1.14.

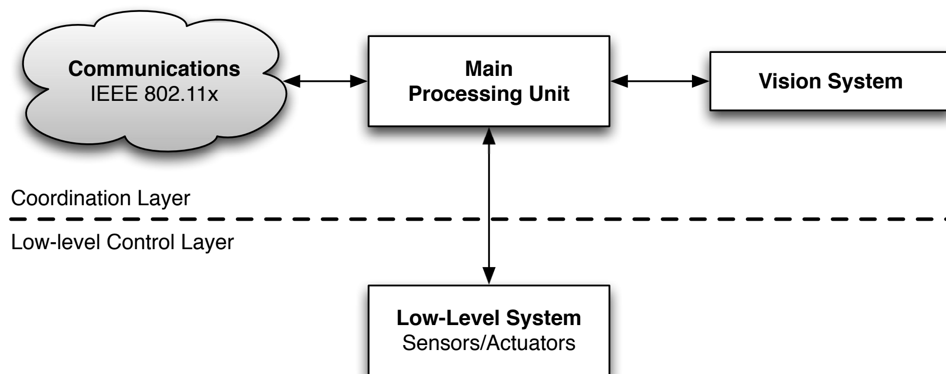


Figure 1.14: Overview of a CAMBADA robot architecture.

Currently, a PC is being used as the main processing unit, responsible for processing the video stream from the vision system, executing the high-level decision, coordination and controlling a series of signals for the low-level system, while reading information coming from other sensors in the platform and communicating with other robots via Wi-Fi IEEE 802.11a or 802.11b standards. This communication protocol supports the **R**eal**t**ime **D**ata **B**ase (RtDB) [9], which is a middleware that provides a seamless access to the complete team state using a distributed database, partially replicated to all team members, as is depicted on Figure 1.15.

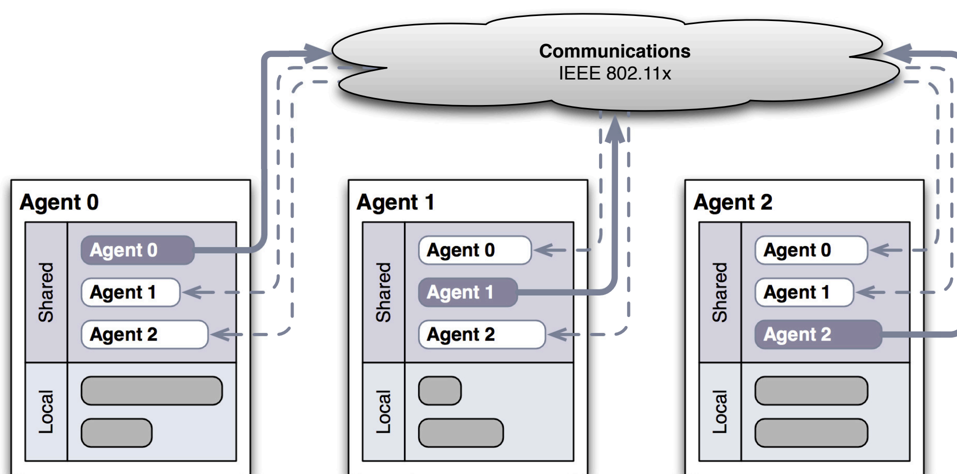


Figure 1.15: A RtDB setup example with three agents.

The information in the RtDB includes the absolute positions and postures of all players, as well as the position of the ball in global coordinates. This is extremely useful while in-game, for example when a certain robot does not see the ball, it can use other teammate's information as a guide. The structure also encompasses some coordination variables, used for the regulation of teamwork tasks.

The RtDB is also intensively used as an inter-process communication mean, due to its capability of defining local memory items that are not broadcasted to other robots, but are still accessible by other processes running on the same agent environment (Figure 1.16, adapted from the Software Description document[2]). This local information includes sensorial data from the vision system and the hardware platform and also commands that are sent to the low-level control layer though a gateway interface.

[2]http://robotica.ua.pt/CAMBADA/docs/qualif2014/CAMBADA-software_structure-2014.pdf

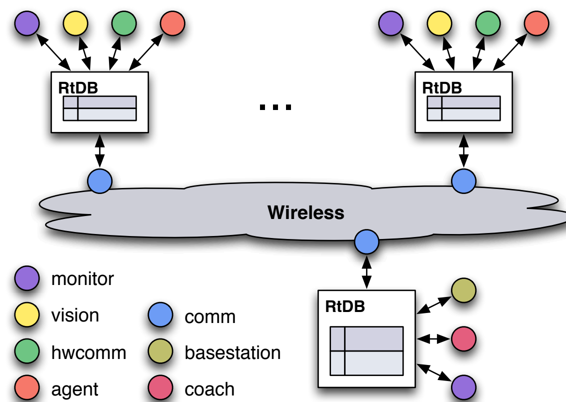


Figure 1.16: Processes running on each robot and on the basestation.

In a nutshell, the description of each process is the following:

- **monitor**
  Controls the other processes running on the same PC. If a process fails, the monitor relaunches it.

- **vision**
  Process responsible for capturing a frame from the digital camera and analyzing it, extracting the relevant information. That data is then stored at a local RtDB item for the agent to use.

- **hwcomm**
  Used to communicate with the low-level layer via a USB gateway.

- **agent**
  The process responsible for decision, coordination and reasoning.

- **comm**
  This process handles the Wi-Fi communication both for sending information to the multicast group and receiving data from other agents [12, 13].

- **basestation**
  An application [14] used to visualize and control the robots state remotely.

- **coach**
  Coach is a software agent responsible for controlling the team basic strategy. It defines formation and assigns robots to specific strategic positions.

1.3.2 Hardware Description

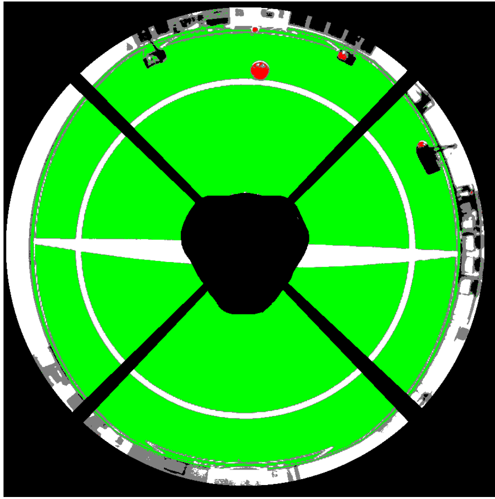
In 2013, the robots' hardware has been fully redesigned and revised. A new platform was completely built in-house with ease of transportation, robustness and modularity in mind, reusing the model and functionalities that have proven to be efficient in the previous platform and introducing new changes in some aspects that required a new approach. Furthermore, a new geometric solution with an asymmetrical hexagon shape was chosen to exploit side dribbling possibilities. In this section, a top-down overview of a CAMBADA robot (Figure 1.17) hardware is made.

Figure 1.17: A CAMBADA Robot

Vision System

The CAMBADA omnidirectional view in Figure 1.18a is obtained by the vision system, which consists of a catadioptric set with an IDS Gigabit Ethernet camera pointing upwards at an hyperbolic mirror (Figure 1.18b). Currently, the video stream is sent to the main processor unit via a wired network at a 50 frames per second (FPS) rate, but the frequency can be adjusted at will (the camera FPS is actually what defines the cycle time/frequency). In the latest version of the vision system, it is even possible to make precise manual adjustments on the position of the mirror in relation to the camera, using some screws on the side.

One of the drawbacks of the previous catadioptric set was its mechanic weakness: the mirror changed its position when a ball hit the set. In the new platform, the mirror is supported by four titanium bars, which is a much stronger solution.



(a) An image acquired by the vision system.

(b) The vision catadioptric set.

Figure 1.18: CAMBADA vision system.

Before actually using it, some of the vision system parameters have to be calibrated, namely: the **camera's colormetric parameters**, the inverse **distance map** computation, the **definition of the region of interest in the image** that has to be processed, and finally the **color ranges** for object segmentation [15].

The parameters of the camera (gain, exposure, gamma, white balance, etc.) can either be calibrated automatically or manually. The distance map consists in a translation from pixel coordinates to real-world relative coordinates at the ground plane. There are several parameters to adjust using a procedure that involves placing the robot manually in the center of the field, recording a video and use an offline application that uses landmarks (either on the robot or the field lines) to extract values of interest via an exploring back-propagation ray-tracing approach and the geometric properties of the mirror. The same application allows the definition of the area of the image that will be processed by the object detection algorithms.

Lastly, the detection of objects is currently a mixture of color-based and shape-based approaches. For the first one, the calibration is done in the HSV (Hue, Saturation and Value) color space. The ranges for each of the objects of interest are specified manually and then translated to a Look-Up Table (LUT) to be used in real-time.

Several radial search lines start on the center of the robot and end in the limits of the image are used to detect objects (field lines, ball, obstacles) by identifying color transitions. Circular search lines are also used to improve the detection. Then, some shape-detection algorithms are applied to increase the precision on the perceived ball position.

Main Processor Unit

The Main Processor Unit consists in a typical laptop (currently a 13" Fujitsu Siemens with an Intel Core i5-3340M CPU at 2.70GHz quad-core processor, 4Gb RAM and a Solid-State Drive), running a lightweight Linux distribution. It is responsible for receiving and analyzing the information coming from the Vision System and sensorial data from the hardware, executing high-level coordination and decision algorithms and producing signals for the Low-Level Layer. This unit is also responsible for the robots inter-communication handling - receiving information coming from the teammates and the base-station and broadcast its own information.

Figure 1.19: CAMBADA Main Processor Unit.

Low-Level Layer

The low-level sensing and actuating system consists in a series of nodes (electronic boards) inter-connected with a CAN (Controlled Area Network) bus. Each board is responsible for certain tasks and the communication between the Main Processor Unit and the Low-Level Bus is done via a USB bi-directional gateway.

Figure 1.20 shows the current configuration of the CAN bus and its nodes, after the changes for the new platform. The Motion function is accomplished by using three different nodes, one per motor/encoder pair. These Motor Nodes are responsible for driving the motors and reading encoder data at the same time. Then, the IMU Node reads information from the Compass, Accelerometer and Gyroscpope sensors. The Kicker Node is responsible for controlling and driving current for the kicker solenoid and controlling the servo-motor

that is used for making passes. Finally, the Grabber Node is responsible for reading the impulses sent by the encoders attached to the grabbers' arms and driving the respective DC motors accordingly.

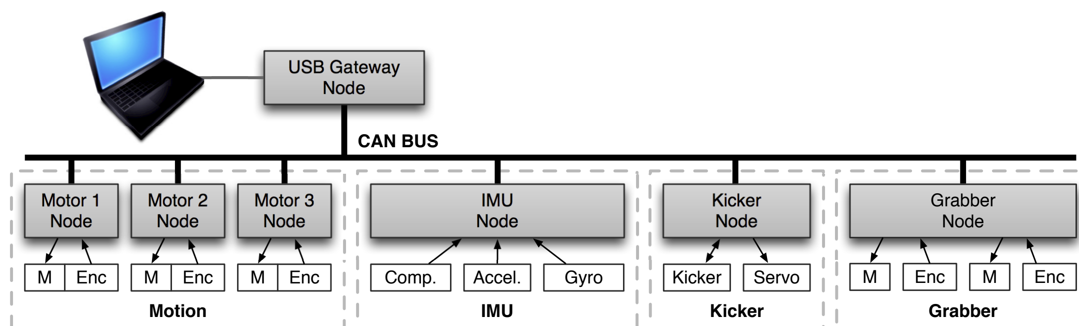


Figure 1.20: Low-level layer architecture.

Ball Handling Mechanism

Another very important mechanical part of the robot is the ball handling mechanism, also known as "grabber" (Figure 1.21). The new grabber is based on a double active handler similar to some of the solutions already presented by other teams. It relies on two DC motors with Sweedish wheels that control the ball's surface velocity on two points. Direction and speed of the ball interface rollers is closed-loop controlled in order to ensure full compliance with the current ball handling rules. It contains two additional free-rolling Sweedish wheels to reduce the friction when dribbling the ball.

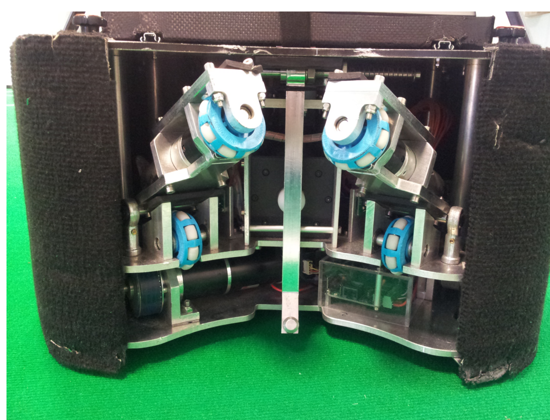


Figure 1.21: The ball handling mechanism.

Ball Kicking Mechanism

The kicker device of the soccer team, responsible for making lob-shots and passes, is an electromagnetic kicker whose main element is an electromechanical solenoid (Figure 1.22). The solenoid consists of a coil, wound around a movable iron core producing a magnetic field when an electric current passes through it.

The magnetic field causes the iron core to move towards the ball, making the lever lift the ball to perform a **lob-shot** (Figure 1.23 and 1.24).

It is also possible to switch to a **pass mode** using a servo-motor that moves the lever sideways (Figure 1.25), making the shaft actuate directly on the ball.

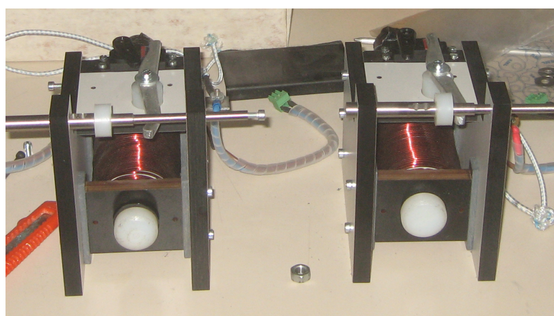


Figure 1.22: The solenoid of the kicker system.

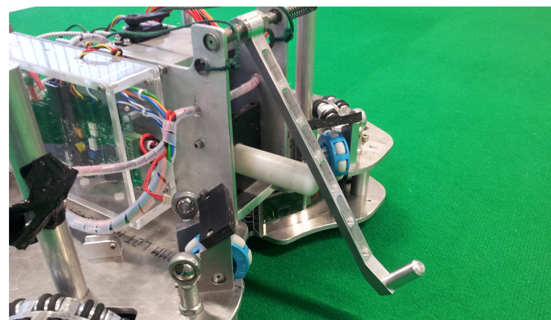


Figure 1.23: The kicker shaft hitting the lever.

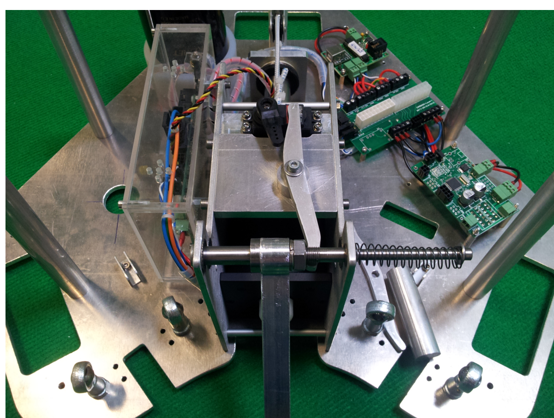


Figure 1.24: Kicker servo-motor in lob-shot mode.

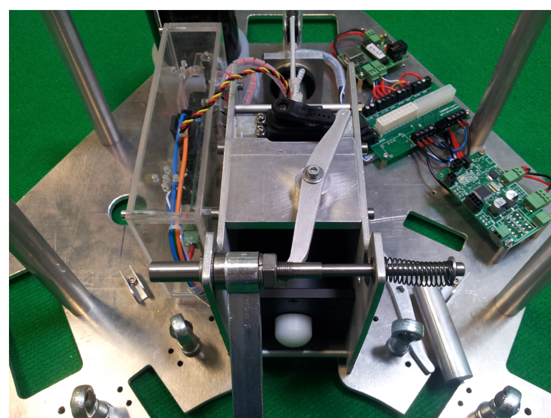


Figure 1.25: Kicker servo-motor in pass mode.

Motion Hardware

In terms of mechanical parts, the wheels that were previously bought were changed to new, custom made, omni-directional wheels based on an aluminum 3-piece sandwich structure, in which 2 sets of 12 off-phase free rollers are supported (Figure 1.26). The concept of the "sweedish wheel" remains, while improving the robustness of the parts, as well as the traction.

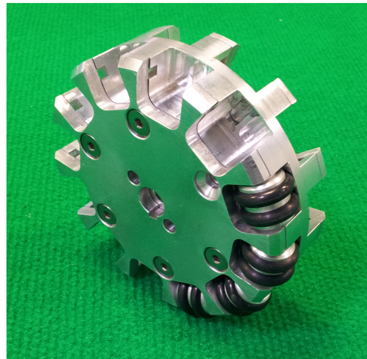


Figure 1.26: One of the wheels partially assembled.

Moreover, a new power transmission system based on synchronous belts and sprockets was introduced in this platform (Figure 1.27). This allowed the team to re-use the current Maxon 150W DC motors providing power transmission to the wheels by a synchronous belt system instead of the previous direct drive approach. The motor control boards had to be upgraded to take into account the changes regarding this new configuration.

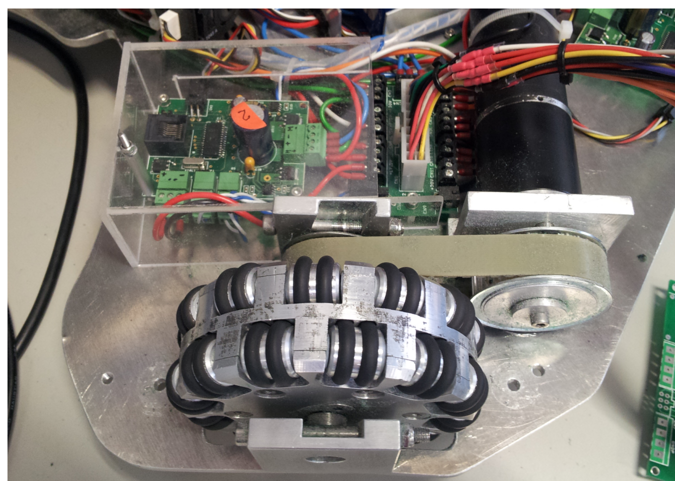


Figure 1.27: Power Transmission System.

1.4 Coordinate Systems

Localization algorithms are able to estimate the position and orientation of a robot, but they work in an absolute base. The CAMBADA team uses two different coordinate systems for different purposes. **Absolute coordinates** are used in respect to the field, where origin resides in the field's center and the y-axis points to the direction where the team is attacking. On the other hand, the **relative coordinate system** has its origin on the center of the robot and its y-axis grows to the front of the robot (Figure 1.28).

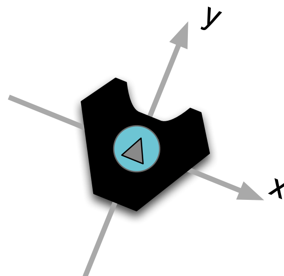


Figure 1.28: The relative coordinate system used by CAMBADA.

1.5 Motivation and Main Contributions

Some improvements on the robots hardware platform (Figure 1.29) have been conducted for the RoboCup 2013 [16] and 2014 [17], allowing the exploitation of new behaviors, such as dribbling with the robot's side.

Figure 1.29: The CAMBADA team robots in RoboCup 2013, in Eindhoven.

In the competitions context, the reasoning capabilities define a success or a failure of a team. Given the high dynamism of the games, it becomes vital to take the correct decisions, at the right time and in a cooperative manner. It is very important that the team is able to make small adjustments between matches as easy and fast as possible, with the minimum risk of compromising the final result. Therefore, the code needs to be extremely clear and flexible. During its 10 years of development, the CAMBADA source code has evolved with contributions from several professors and students and like any other long-term project, it was requiring a refactoring. With the changes on the hardware platform, this was the perfect timing for the team to make enhancements also on the software level.

This thesis intends to improve the structure of the agent, from the code organization to the actual software architecture, while streamlining the whole process of developing, testing, debugging and refining agent behaviors. In brief, the developed work included the improvement of the current CAMBADA agent architecture, by presenting a new solution.

The constant evolution of the MSL pushes teams to adapt to new rules each new year. In this context, some novel behaviors were developed and others have been refined in the new software architecture. Moreover, for the creation, test and validation of these behaviors, a series of applications were needed for development, calibration and debugging, namely an upgrade to the basestation visualization widget and an Augmented Reality tool that is useful, not only for visualization purposes, but is also helpful for development of behaviors that are dependent on the opponents team dynamic attitude. All the developed work during in thesis was not only fully integrated into the main CAMBADA source-code, but also was it used and presented in the IranOpen 2014 [18] and the RoboCup 2014 [17].

1.6 Thesis Structure

In chapter 2, an overview on robotics agents architecture is presented and a simple introduction on Multi-Agent Systems is performed. The chapter finalizes by summarizing the CAMBADA general architecture with main focus on the software agent architecture. Following, in chapter 3 the new Agent Architecture is defined and explained in detail. An example is given for one of the main roles in free-play, the Striker Role. Moreover, a new class for defining HeightMaps is presented. Chapter 4 explain a series of new and improved behaviors, including ball pass, reception, dribbling, kicking and goal protection. In order to develop, test and debug these new behaviors, a series of tools have also been developed and presented in chapter 5. Finally, a conclusion is made in Chapter 6 on all the developed work.

Chapter 2

Robotic Agent Architectures

This chapter introduces some concepts needed to understand the architectures of robotic agents. The definition of Software Agent is presented and the bridge to Multi-Agent Systems is made. Finally, an analysis on coordination and decision is presented in the context of the Middle-Size League, with special focus on the CAMBADA Agent Architecture at the time this thesis started.

2.1 Software Agent Architectures

A Software Agent is, by definition, a complex software entity in the form of a computer program, which works towards a generic high-level goal, as opposed to discrete tasks. It works autonomously and continuously in a dynamic environment, making decisions based on what it perceives to be the current world state and taking actions that may influence the world [19].

Software Agents usually include abilities such as learning and reasoning, and are able to transform goals into action tasks without continuous direct supervision or control. Agent architectures can be organized mainly in four different groups:

- **Reactive Agent**

  Reactive agents base their decisions on the present information, discarding all references to history. In this type of architecture, there is no world-model representation and the final behavior can be obtained with very little processing power, because there are no internal states and the agent output are simple deterministic reflexes to the perceptual inputs. An example of this type of agent is Brook's subsumption architecture [20].

- **Deliberative Agent**

  Usually, the information perceived by an agent is insufficient to make a smart decision, because it requires a broader world understanding. Under those conditions, the agent needs to retain some information as an internal state. The deliberative agent, also known as intentional or hierarchical agent, has a good symbolic world model, can build symbolic maps and has planning abilities, which it uses to make decisions like selecting behaviors [21, 22].

- **Behavior-based Agent**

  In this architecture, an agent is decomposed into simple behaviors, each of them storing their own world representation of the world. The final behavior is chosen by evaluating a set of pre-defined conditions. This modular approach falls between the reactive and deliberative agents and simplifies the development of new behaviors by enabling fast prototyping. Pattie Maes [23] presented some advantages and disadvantages of this method.

- **Hybrid Agent**

  The hybrid agent combines the ideas of the reactive and deliberative approaches, by trying to get the best out of both. In this architecture, symbolic knowledge of the world state is very important, because all the decisions are based upon this representation. Planning and immediate reaction are evaluated in parallel, so the challenge of this approach is to join and coordinate both [24, 25].

2.2 Multi-Agent Systems

> *"Multi-Agent Systems (MAS) is a subfield of AI that aims to provide both principles for construction of complex systems involving multiple agents and mechanisms for coordination of independent agents' behaviors."* [26]

MAS [27] enable the use of multiple problem solving agents (which may have their own interests and goals and may differ from agent to agent) collaborating with each other to achieve a common goal. Although not all of the problems need to be tackled with MAS, Peter Stone highlights a series of advantages for using them in his survey [26], and some of them can easily be related to a robotics soccer team participating in the Middle-Size League:

- **Some domains require it**
  That is exactly the case of an MSL team. A strong team can only be developed with multiple robots, each playing a specific role in the game.

- **Parallelism**
  The ability to have more than one agent working in parallel could potentially speed up a process. In particular, if the position of the ball in the field is unknown, a smart way to search it may be to use all the robots instead of just one.

- **Robustness**
  If one agent fails, we can replace it with another agent probably faster than we can repair the first. In the case of a soccer match, it is sometimes needed to take a robot out of the field when it has some problem. If it is the case of a robot responsible for restarting the game in a set-piece situation, another robot has to immediately take over its position.

- **Simpler Programming**
  From an agent's perspective, it may be simpler to identify subtasks and assign them to different agents than to tackle the whole task with a centralized agent.

2.2.1 MAS Example Applications

Agogino and Tumer presented a "multi-agent approach to managing air traffic flow" [28]. The developed algorithm included agents that used reinforcement learning to reduce congestion through local actions. A simulator developed by NASA (FACET, in Figure 2.1) was used to measure the efficiency of their approach, showing that agents receiving personalized rewards reduce congestion by up to 80% over agents receiving a global reward and by up to 90% over a current industry approach.

RoboCup Rescue simulation (Figure 2.2) also requires the use of pure Multi-Agent solutions for simulating search and rescue operations in large-scale disasters. In this competition, several heterogeneous agents (Police, Fire Brigades, and Ambulances) have to cooperate in order to save as many civilian as possible and minimize the damage of the simulated city disaster scenario.

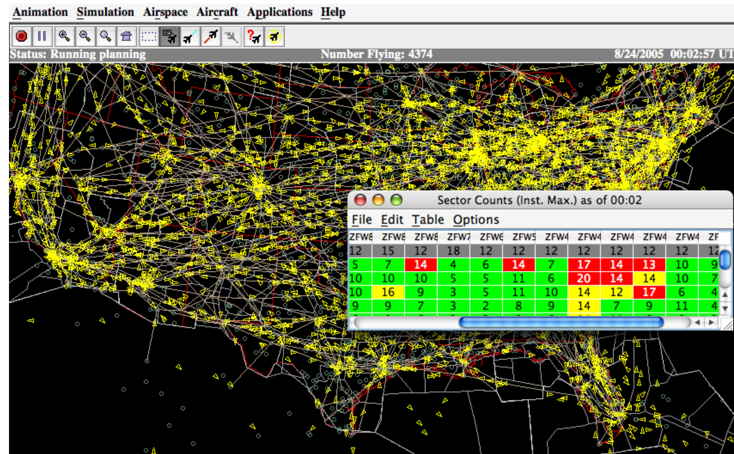


Figure 2.1: FACET screenshot displaying traffic routes and air flow statistics.

Figure 2.2: An example of a rescue situation in a simulated city.

2.3 Coordination and Decision in MSL

In the Middle-Size League there is a huge room for autonomy in each team implementation on the coordination and decision layer of the agents. While some teams rely on case-based reasoning, others use more sophisticated methods. In this section, the approaches of some of the most successful international teams are detailed.

In 2012, Tech United Eindhoven from The Netherlands presented their utility field approach for dynamic positioning on the Scientific Challenge. Their source-code is publicly available and prove that they are currently using a mixture of Finite State Machines and Utility Fields for the strategy and decision.

In RoboCup 2013 Scientific Challenge, MRL team from Qazvin University, Iran, presented their Integrated Development Environment which included a template wizard for

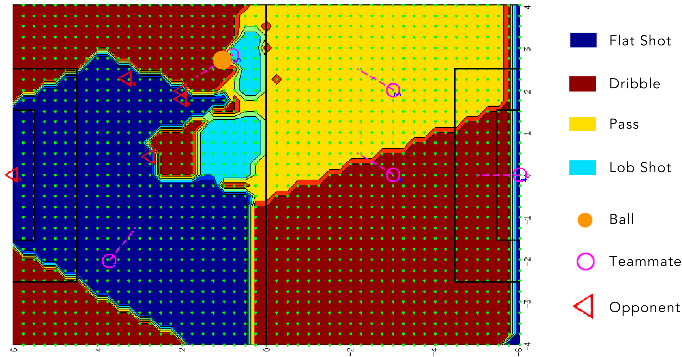


Figure 2.3: Tech United task utility field, adapted from their 2014 Team Description Paper [29].

---

creation of new states in their Finite State Machines. They rely purely on case-based reasoning, supported by some useful graphical representations of the transitions.

The Carpe Noctem team from Kassel, Germany, developed a framework: ALICA (A Language for Interactive Cooperative Agents) [30]. It is a planning language for Multi-Agent Systems that supports hierarchical plans, agent heterogeneity and explicit synchronization between team members.

The Brainstormers Tribots team from Freiburg University were one of the more successful teams in the RoboCup championship. They had a Behavior-Based approach [31, 32], which combined the concepts of two different architectures: Brooks' subsumption architecture [20] and Rao&Georgeff's Belief-Desire-Intention architecture [33].

The CoPS team from University of Stuttgart used a State Machine modeled in XABSL [34]. The XABSL file was generated using a special GUI editor, where the pre-conditions for each state could easily be edited.

2.4 CAMBADA Agent Architecture in 2013

At the time of the start of this thesis, the CAMBADA agent architecture was behavior-based, with Finite State Machines for decision and some real-time adaptive dynamism. The base strategy was defined beforehand, but minor adjustments were made in gameplay based on the opponent's team attitude. The main blocks of the architecture are presented in this section.

1. `Behaviors`

   Responsible for executing certain basic tasks, like moving to a point (`Move`), dribbling

(`Dribble`), kicking the ball (`Kick`), receiving a pass (`CatchBall`) or interception the ball (`Active Interception`). Behaviors take some input (typically a target position and/or orientation) in the constructor and have a `calculate()` method to compute the required velocities in X, Y and Angular components, typically using (although not restricted to) three PID compensators, one for each component.

2. `Roles`
   Responsible for assuming a certain role in the game, such as `Striker`, `Midfielder` or `Goalie` for instance. The `Roles` use behaviors to achieve their objectives and were implemented as Finite State Machines, although the interface allowed the use of other behavior selection techniques.

3. `Decision`
   A unique module that is responsible for selecting the `Role`, based on the current game conditions and some history. As a rule of thumb, robot number 1 is always the `Goalie`. Then, in free-play, the robot closer to the ball is the `Striker`, the remain are `Midfielders`. On the other hand, if we are in an own set-piece, the `Striker` becomes the `Replacer` (the robot responsible to perform the pass) and the others are `Receivers`. If the set-piece is for the opponent team, all the robots assume the `Barrier` Role, that tries to block opponent's passes.

4. `Strategy`
   Responsible of defining the team's strategic positioning, based on the current game state, usually based on the ball position. Different formations can be defined beforehand, using Delaunay Triangulation as proposed for the first time in the simulation league [35]. The coach process, running on the basestation computer, decides which formation to use at a given time, given certain pre-defined conditions [36].

5. `Integrator`
   The module responsible for gathering information from the sensors, filtering it and updating the agent's internal state of the world.

6. `WorldState`
   A module that holds information about the world state, such as robots' position, velocity, role, behavior, perceived ball position and velocity by each robot, some coordination flags, lines or vectors, obstacles pose, among other information.

This certainly is a well organized approach. However, there were two major drawbacks of this architecture, that were preventing the team from evolving faster:

- **Lack of History** - at each new cycle, the Decision module deletes the last picked role and instantiates a new one, based a set of conditions. Then the Role has its own conditions to instantiate a new Behavior. All of this reasoning was lost for the next cycle.

- **Complexity of Roles** - Roles were implemented as Finite State Machines (FSM) that needed to handle several different aspects of the gameplay, including several special conditions, with multiple control variables, `Timers` and more. Furthermore, taking the example of when the robot has the ball engaged, it has mainly three possible Behaviors (`Dribble`, `Pass` or `Kick`), which can be invoked with different parameters, therefore, as expected, all the complexity is moved to the module calling these Behaviors - the Role. The Role decides the parameters to input to the selected Behavior to achieve a certain result. This helps explaining why most of the roles had hundreds of lines of code (between 176 and 675) and making simple changes was always a challenge.

The main goal of this thesis is to improve the structure of the software agent code, tackling the above stated problems: on one hand, add persistence properties to Behaviors and Roles, so that history can influence the decisions and, on the other hand, try to reduce the complexity of roles. Chapter 3 introduces the proposed software agent architecture in detail.

Chapter 3

CAMBADA Agent Architecture

To address some of the problems identified in the previous agent architecture, one of the objectives of this thesis was the development of a new one. The first step was to identify the various modules that, when combined, would lead to a strategy solution. The refactoring was mainly aimed at the Behaviors, Roles and Decision layers, but some adjustments have also been made on the Worldstate. Figure 3.1 compares the main blocks of the proposed CAMBADA software agent architecture with the previous one. The Roles described in Section 3.5 contain an Arbitrator (Section 3.4) to select a Behavior (Section 3.3) that then calls a certain Controller (Section 3.2) to calculate a DriveVector (Section 3.1) with all the required low-level information.

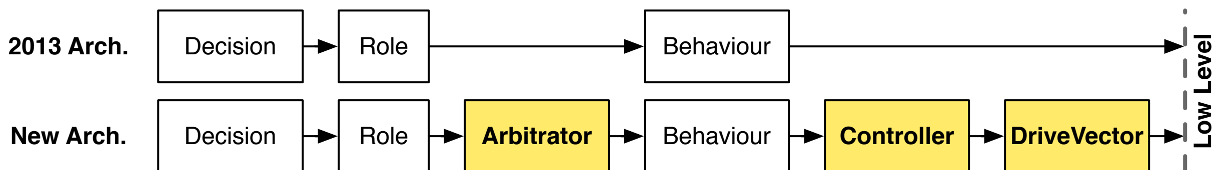


Figure 3.1: Comparison between the old and new software agent architectures.

The Arbitrator contains a set of Behaviors instantiated in the initialization phase, thus eliminating the problems of history loss. At each cycle, it selects one of them using a set of conditions, as an alternative to Finite State Machines. This architecture allows both the Behaviors and the Roles to hold a persistent internal state throughout the agent lifespan. The Controller deals with the generation of robot velocities and currently implements a PID compensator, while leaving space for the exploitation of other types of compensators in the future, transversal to any Behavior. Therefore, the Behaviors are responsible for calculating targets to input into a Controller that knows the system and how to control it.

3.1 DriveVector

As last step, at each cycle, the agent needs to calculate some information to send to the low-level - a `DriveVector`. Although the name chosen may be misleading, the `DriveVector` not only contains the desired velocities, but also some data for the kicker and grabber desired states. This module contains the following items:

- `float velX`
  The linear velocity in the robot's xx axis (in m/s).
- `float velY`
  The linear velocity in the robot's yy axis (in m/s).
- `float velA`
  The angular velocity around robot's center (in rad/s).
- `unsigned char kickPower`
  The requested power to the kicker. Ranges from 0 (no kick) 50 (maximum strength). The Most Significant Bit (MSB) in the byte toggles between a pass and a lob-shot.
- `GrabberType grabber`
  One of the following: `GRABBER_ON`, `GRABBER_OFF` or `GRABBER_DEFAULT`.

The `kickPower` is zero by default, and there are two methods responsible for calculating the correct kickPower for a certain distance regarding a lob-shot or a pass. As it is a public member of the `DriveVector` class, it is still possible to set its value manually, although it is not recommended.

The grabber is set to `GRABBER_DEFAULT` at the beginning of each new cycle and if it is not changed by any `Behavior` or `Role`, a method is called that turns it ON under certain conditions (mainly the distance and alignment of the robot regarding the ball). This approach enables the use of a default grabber state, but also the possibility of forcing it to a certain state at will.

Currently, a unique `DriveVector` object exists during the agent's lifespan. It is completely orthogonal to the various decision layers, in order to extend the flexibility of the architecture. Therefore, it can be easily changed by the `Controllers`, `Behaviors`, `Roles`, `Decision`, and finally in the `Cambada` class where the "main" cycle block resides. This means that a higher level block can override decisions made on layers below. However, the architecture allows the creation of multiple DriveVectors during the decision cycle, but ultimately, a single one is sent to the low-level.

3.1.1 Interface

A few methods have been added to this class, in order to have a common way of manipulate some of the inner attributes:

- `void allOff()`
  Turns all of the attributes off, i.e. all velocities and the kick power are set to zero and the grabber is turned off.

- `void motorsOff()`
  Sets all velocities to zero.

- `void pass( double distance )`
  Calculates and sets a value for the kickPower attribute as a pass strength based on the distance of the pass (including the set of the MSB).

- `void kick( double distance, double height )`
  Calculates and sets a value for the kickPower attribute as a kick strength based on the distance of the kick and the desired ball height at that distance.

- `void grabberControl()`
  This method defines the default behavior of the ball handling mechanism. If no role or behavior sets a value for the grabber attribute, this function is called and, based on certain conditions, sets or resets the grabber state.

- `void setLinearVel( float absLinearVel )`
  Scales the current linear velocity to a module specified by the float parameter, maintaining its orientation.

- `void limitVel( float maxLinVel, float maxAngVel )`
  Clips the linear and angular velocities to the specified values.

3.2 Controllers

The desired robot movement determines the linear and the angular velocities to be applied. These values were being calculated inside the Behaviors, with most of them in the form of PID controllers. However, in the future there might arise the interest in exploring other types of controllers. This was how the concept of `Controller` was born in the CAMBADA agent context. `Controllers` are the modules responsible for calculating

the desired robot velocities, based on a certain target which may vary from controller to controller.

Controllers can be (and are) used by the behaviors intensively, eliminating the need of calling the PID compensator individually on each behavior.

Therefore, in order to achieve different type of movements, four types of controllers were implemented as described next.

3.2.1 CMove

This controller is used for straight line movements and positioning. It controls both position and orientation of the robot and allows the user to avoid obstacles and set a top speed if needed.

- `void setAvoidLevel( avoidLevel avoid )`
  Allows the user to set an avoidance level (no avoidance, avoid ball only, avoid obstacles, etc.) before calculating the output DriveVector with the `calcVel` method.

- `void setDistribute( bool distribute )`
  Setting `distribute` to `true` will enable the controller to distribute the orientation along the linear movement. This will result in both corrections ending at the same time. If the movement is not distributed, the linear and angular PIDs are called independently and as a result, the angular movement takes precedence since it can compensate much faster than the linear one.

- `void calcVel(DriveVector* dv, Vec relPos, Vec relOri, float maxSpeed)`
  The method is used to calculate the output DriveVector, based on a relative position and relative orientation targets. It is even possible to limit the linear velocity using the maxSpeed parameter.

3.2.2 CArc

The CArc controller implements the ideas coming from the previous Dribble behavior, which results in arc movements. Receives an orientation target and a maximum linear velocity as parameters for calculations.

- `void calcVel(DriveVector* dv, float relOri, float maxSpeed)`
  Calculates the output velocities in a `DriveVector`, based on the given relative orientation target and on a maximum linear velocity.

3.2.3 CRotateAroundTheBall

The result of this `Controller` is a set of velocities that make a robot rotate around a point that is the ball center when it is fully engaged. It is intended to be used when the robot has possession of the ball and wants to rotate around it.

- `void calcVel(DriveVector* dv, float relOri)`
  Calculates the output velocities in a `DriveVector`, based on the given orientation target. The velocity is the fastest possible in order for the robot not to lose the ball.

3.2.4 CRotate

This controller results in a simple rotation around the robot center. The only required parameter is the relative angular target.

- `void calcVel(DriveVector* dv, float relOri)`
  Calculates the output velocities in a `DriveVector`, based on the given orientation target. The output linear velocity is zero and the angular velocity is calculated from the angular PID.

3.3 Behaviors

In the 1st International RoboCup MSL Workshop [1], held in Kassel in November 2008, the Brainstormers Tribots (Neuroinfromatics Group, from University of Osnabrück) presented their Behavior-Based approach [31, 32], which combined the concepts of two different architectures: Brooks' subsumption architecture [20] and Rao&Georgeff's Belief-Desire-Intention architecture [33]. Their implementation included an interface extension to the behaviors, with two additional conditions that reflect the respective constraints:

- **Invocation Condition** (IC) - specifies the conditions that will trigger the behavior.
- **Commitment Condition** (CC) - reflects the conditions for the behavior to keep in control.

Linking `Behaviors` to the respective activation and release conditions seemed like a good approach, therefore the `Behavior` interface has been extended to have, not only a `calculate()` method, but also two important methods to verify **Invocation Conditions**

[1] `http://carpenoctem.das-lab.net/research/1stRMW`, accessed in 1 June 2014.

(IC) and **Commitment Conditions** (CC). The behavior can be selected if the IC are true and the CC will remain true while that behavior is suitable for the situation.

Moving from the previous architecture to this one included the formulation of more complex behaviors (that previously were states of the FSM) and also the extraction of all the transition conditions and special pre-conditions associated with that state to formulate the IC and CC.

3.3.1 Interface

The new Behavior base class has several methods that have to be implemented by the derived behaviors.

- `virtual void calculate(DriveVector* dv)`
  The main block of the Behaviors, where it calculates a DriveVector with the desired velocities and the kicker and grabber states.

- `virtual bool checkIC()`
  The derived Behaviors have to implement this method and define the Invocation Conditions inside.

- `virtual bool checkCC()`
  The derived Behaviors have to implement this method and define the Commitment Conditions inside.

- `virtual void gainControl()`
  The derived Behaviors can implement this method to be notified when they gain control. The callback can be used, for instance, to initialize variables.

- `virtual void loseControl()`
  The derived Behaviors can implement this method to be notified when they lose control to other Behavior. The callback can be used, for instance, to clean up variables.

3.3.2 List of Current Behaviors

After the refactoring work, a list of behaviors was produced and is presented in this section. Some new behaviors have been developed and integrated into this new architecture as part of the developed work for this thesis. Those were not included in this list, since they will be explained in detail in Chapter 4.

Generic Behaviors

- **BAvoidTheirGoalArea** makes the robot avoid the opponent's goal area. If the robot is going to move inside that area, this behavior will be triggered and force the robot to move outside the area, following the target.

- **BStop** makes the robot stop and it is used as a default or fallback behavior. The IC and CC are always true.

- **BStopRobotGS** inherits the BStop behavior, overriding the IC and CC to be true only when the game is in a STOP state.

Ball Handling Behaviors

- **BKickToTheirGoal** makes the robot rotate to the opponent's goal and kick the ball, under certain conditions.

- **BRelieveBall** is used to kick the ball to the opponent's side as fast as possible, usually as a fallback behavior.

- **BDribble** makes the robot dribble the ball towards the opponent goal.

Behaviors Without Ball

- **BActiveInterception** is used to grab a free-rolling ball, taking into account the acceleration and maximum velocity of the robot.

- **BCatchBall** is used to catch the ball from a team-mate pass.

- **BGoToBall** is a Priority Arbitrator used to grab the ball or search for it in case its position is unknown.

- **BGoToVisibleBall** goes directly to the ball to grab it.

- **BSearchBall** searches the ball when its position is unknown.

Striker Behaviors

- **BStrikerGoToBall** inherits the BGoToBall and adds some preconditions for the Striker.

- **BStrikerOurFieldDribble** was used to dribble in our side of the field, when the ball was grabbed in our side. This was used because in the 2013 rules, no team was allowed to dribble the ball from their side to the opponent's side of the field.

- **BStrikerPass** is used to make a pass to a teammate.

- **BStrikerRelieveBall** inherits BRelieveBall, adding preconditions for when there are no receivers in free-play or the robot is taking too long dribbling on the own side of the field.

Midfielder Behaviors

- **BGoToStrategicPosition** makes the midfielder to position itself in its strategic position.

- **BMidLineBlock** makes the midfielder block a possible pass over the field mid-line.

- **BMidfielderReceiveBall** used to receive a ball in free-play.

Goal-Keeper Behaviors

- **BGoalieDefend** implements the basic goal-keeper defensive behavior.

- **BGoalieGoToGoal** is triggered to move the robot to the center of their own goal when it is outside certain limits.

- **BGoalieKickAway** is used to grab the ball when it is stopped for some time in the penalty area and kick it away.

Replacer Behaviors

- **BReplacerAlign** aligns the robot with the target receiver.

- **BReplacerBallPassedStop** stops the robot after making a pass.

- **BReplacerPass** makes a pass to a receiver.

- **BReplacerPos** positions the Replacer near the ball before the referee starts the play.

3.4 Arbitrator

As stated before, Finite State Machines (FSM) proved themselves not to be the best option in this scenario, so a research has been made through other implementations. In the context of MSL, the dynamic environment forces teams to develop an hybrid agent: a reactive component is mandatory to be able to react to fast changes of the environment (the opponent team), but some deliberative part is still needed, since there is a growing need of predicting the near future to base decisions on. The concept of `Arbitrator` was introduced as the module responsible for selecting a certain behavior in a cycle, based on the ideas of the Tribots [31]. Several `Behaviors` can be added to an `Arbitrator` as options and are selected based on their IC and CC conditions.

3.4.1 Priority Arbitrator

The Priority Arbitrator receives a set of candidate behaviors, which are arranged in a priority queue (in the `options` field). In run-time, it will run the Algorithm 1 to select the best `Behavior` among all options.

Algorithm 1 Priority Arbitrator

```
Input: currentBehavior, options          ▷ The last picked behavior and the options list
Output: currentBehavior, driveVector     ▷ The selected behavior and its output

if not currentBehavior.CC() then         ▷ If the current behavior is not suitable
    currentBehavior ← BStop              ▷ Stop the robot, by default
end if
for i = 0 to options.length() do         ▷ Iterate through all options
    if options[i] = currentBehavior then ▷ If this is the current behavior
        break                            ▷ continue with the same behavior
    end if
    if options[i].IC() then              ▷ If the IC is true
        currentBehavior ← options[i]     ▷ Select this behavior
        break
    end if
end for
driveVector ← currentBehavior.calculate()
return driveVector
```

Figure 3.2 shows an example of the Priority Arbitrator functionality with 4 `Behavior` candidates (A, B, C and D) in five consecutive cycles. The chosen behavior (amongst the options) at each cycle is highlighted.

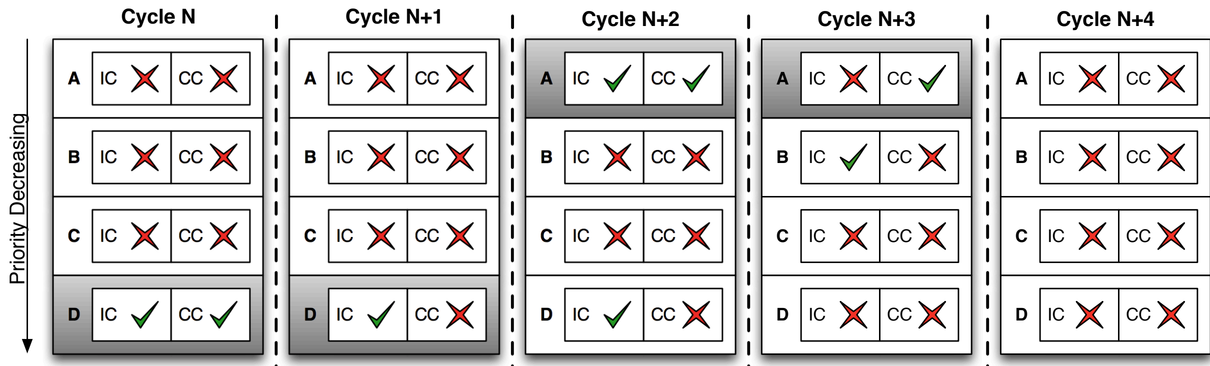


Figure 3.2: Example of the Priority Arbitrator.

In the first cycle, the first behavior (from top to bottom) with a true IC becomes the active behavior. In Cycle N+1, its CC becomes false, but it is still chosen as the best behavior, since its IC is still true. In the next cycle, a behavior with higher priority has a positive IC and becomes the active behavior. In Cycle N+3, the IC of behavior A becomes false, but since its CC is still true, it is the chosen behavior for that cycle. Lastly, in cycle N+4, the CC of behavior A became false and none of the options has a positive IC, therefore the fallback behavior is activated (`BStop`), which stops the robot.

The IC and CC conditions typically depend on certain values and the decision is done by comparing that values with a certain threshold. When using the same threshold for both IC and CC, it is easy to end up with a cyclic unstable behavior.

For example, a `Behavior` has both IC and CC that return true when the ball is on the opponent side of the field and the ball is on the middle line. Due to camera sensor noise and errors in measurements, from cycle to cycle, the ball may easily be interpreted to be alternating between our side and the opponent side. This results in an unstable result, because that `Behavior` keeps activating and releasing.

Therefore, having different thresholds for a behavior's IC and CC becomes very convenient and this is a procedure that is being used to create hysteresis and to overcome problems like the one stated above.

3.4.2 Finish Plan First Arbitrator

A "finish plan first" arbitrator takes a behavior and does not release it until its CC becomes false. Consequently, each `Behavior` has to execute its task until the end, which is defined by the CC. The algorithm becomes a little bit different and is presented in Algorithm 2.

Algorithm 2 Finish Plan First

```
Input: currentBehavior, options            ▷ The last picked behavior and the options list
Output: currentBehavior, driveVector       ▷ The selected behavior and its output

if not currentBehavior.CC() then           ▷ If the current behavior is not suitable
    currentBehavior ← BStop                ▷ Stop the robot, by default
    for i = 0 to options.length() do       ▷ Iterate through all options
        if options[i].IC() then            ▷ If the IC is true
            currentBehavior ← options[i]   ▷ Select this behavior
            break
        end if
    end for
end if
driveVector ← currentBehavior.calculate()
return driveVector
```

This type of arbitrator has a huge importance, since it helps improving the stability. For instance, several behaviors are available to approach the ball in a different manner and as soon as one is selected, keeping switching between "modes" is perhaps not desired. To overcome this problem, this type of arbitration can be used to select one behavior once and let it keep control until it believes its task is achieved or is not achievable.

3.4.3 Combining Both Arbitrator Types

Having set the rules and fundamentals of the two types of `Arbitrators`, there was the need of mixing the ideas of both, because sometimes a Priority Arbitrator is not suitable and the Finish Plan First Arbitrator can become limitative in some cases. These two support the final version of the CAMBADA `Arbitrator` (algorithm 3), where a level of arbitration can be set per `Behavior`. This property is set on the assignment of the `Behaviors` on an Arbitrator. As a result, an `Arbitrator` is a Priority one by default, but

can act as a Finish Plan First for certain `Behaviors`.

When a Behavior is added to an `Arbitrator`, it can be flagged as **non-interruptible**. This means that even if a higher priority `Behavior` becomes positive in its IC, the arbitrator will not drop the non-interruptible `Behavior`.

Algorithm 3 CAMBADA Arbitrator

Input: currentBehavior, options ▷ The last picked behavior and the options list
Output: currentBehavior, driveVector ▷ The selected behavior and its output

```
if not currentBehavior.CC() then                  ▷ If the current behavior is not suitable
    currentBehavior ← BStop                       ▷ Stop the robot, by default
    if currentBehavior.nonInterruptable then
        for i = 0 to options.length() do          ▷ Iterate through all options
            if options[i].IC() then               ▷ If the IC is true
                currentBehavior ← options[i]      ▷ Select this behavior
                break
            end if
        end for
    end if
end if
if not currentBehavior.nonInterruptable then
    for i = 0 to options.length() do              ▷ Iterate through all options
        if options[i] = currentBehavior then      ▷ If this is the current behavior
            break                                 ▷ continue with the same behavior
        end if
        if options[i].IC() then                   ▷ If the IC is true
            currentBehavior ← options[i]          ▷ Select this behavior
            break
        end if
    end for
end if
driveVector ← currentBehavior.calculate()
return driveVector
```

3.4.4 Inheritance

One interesting feature of the `Arbitrators` is that they are `Behaviors` themselves, however, instead of calculating a `DriveVector`, they are responsible for calling other behaviors. This is extremely useful for general purpose behaviors, like "go to the ball". The way the robot approaches the ball depends on several conditions, so a generic `BGoToBall` behavior was developed, which is a `Arbitrator` itself that will evaluate the IC and CC of various types of movements to grab the ball.

3.5 Roles

Roles now have an `Arbitrator` by default, as well as a callback method that can be implemented by the derived `Roles` to make decisions and/or update internal variables or own worldstate information. In the constructor of the `Role`, a set of behaviors is added to its `Arbitrator` queue. Behaviors and Roles are created once in the beginning and are selected based on their conditions, and can then keep track of the past decisions.

3.5.1 The Role Striker Example

Figure 3.3 shows a graphical representation of the behaviors currently in use for the Striker.

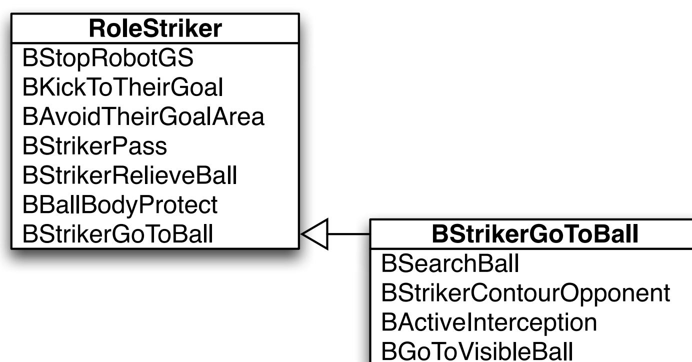


Figure 3.3: Behaviors added for the RoleStriker.

The most important thing above all the possible artificial intelligence is safety. On the top of the behaviors list is `BStopRobotGS`, which will be triggered whenever the game state changes to **Stop**. Although the Decision module releases the `RoleStriker` and activates

the RoleStop under these conditions, this module is overridden if a Role is forced using the basestation.

BKickToTheirGoal appears next on the list, since the most important objective in a soccer game is to score. Then, it is also important to avoid the opponents goal area (BAvoidTheirGoalArea).

The behavior used for passing the ball in free-play is BStrikerPass. It is activated when the robot is aligned with a receiver Midfielder reporting a clear line of sight.

Next on the list is BStrikerRelieveBall, which is triggered when the Striker is playing alone and has no receiver to pass the ball to.

The BBallBodyProtect behavior is explained in detail in section 4.2 and it is responsible for protecting the ball and for the the alignment with possible receivers.

Lastly on the RoleStriker Arbitrator options list is BStrikerGoToBall, which is a Priority Arbitrator itself, with four different behaviors to choose from. This arbitrator is composed of BSearchBall that searches for the ball in case this is not seen by any of the team mate robots, BStrikerContourOpponent that is explained in section 4.5, BActiveInterception, that is used to approach a ball that is not being dribbled by a robot, and finally BGoToVisibleBall that goes directly to the ball to grab it.

3.6 HeightMap Class

Some of the developed behaviors (described in Chapter 4) rely on spacial height maps (or cost maps). The TCOD[2] library was used, since it has built-in toolkits for management of height maps and field of view calculations.

To facilitate the interface with these height maps, a HeightMap class has been developed. Internally, it contains a TCOD height map, but provides several methods for easy manipulation of the map using intuitive parameters, since all the calculations regarding the transformations between grid and world coordinates are performed seamlessly.

3.6.1 Interface

In this section, a list of the provided public methods is presented.

- `void clear()`
  Clears the map, reseting all cells to zero.

[2]http://doryen.eptalys.net/libtcod

- `geom::Vec grid2world(int x,int y)`
  Conversion from grid coordinates to world coordinates.

- `void world2grid(geom::Vec pos, int &px, int &py)`
  Conversion from world coordinates to grid coordinates.

- `void addHill(geom::Vec point, float radius, float height)`
  Adds a hill centered on the position defined by the parameter `point` with a certain radius and height.

- `void digHill(geom::Vec point, float radius, float height)`
  Digs a hill centered on the position defined by the parameter `point` with a certain radius and height. Digging provides a way of limiting the height. Digging twice over a zone that was previously digged, makes no difference on the map, whereas adding two hills has a cumulative effect.

- `void addOffset(float value)`
  The value is added to every cell in the whole map.

- `void addOffset(geom::Circle circle, float value, bool inside)`
  The value is added to the cells inside or outside the circle, depending on the last parameter.

- `void addOffset(geom::XYRectangle rect, float value, bool inside)`
  The value is added to the cells inside or outside the rectangle, depending on the last parameter.

- `void addOffset(geom::Line line, float value, bool right)`
  The value is added to the cells on the right side or left side of the line, depending on the last parameter.

- `void add(HeightMap* map2)`
  The cell values of map2 are added to the inner height map.

- `void cloneTo(HeightMap* destination)`
  The cell values of the inner height map are copied to the destination HeightMap.

- `geom::Vec getMaxPos()`
  Returns the position of the cell with the highest value in world coordinates.

- `geom::Vec getMinPos()`
  Returns the position of the cell with the lowest value in world coordinates.

- `float getMaxVal()`
  Returns the highest cell value on the map.

- `float getMinVal()`
  Returns the lowest cell value on the map.

- `float getVal(geom::Vec pos)`
  Returns the cell value from a defined position.

- `void normalize(float valMin = 0.0, float valMax = 1.0)`
  Normalizes the map between two defined values (0.0 and 1.0, by default).

These are part of the environment symbolic representation and thus were included in the WorldState module, to be easily accessible from any Behavior or Role.

3.6.2 Cost Maps

With the developed `HeightMap` class, several different spacial cost maps have been implemented in the `WorldState`: `fovBall` (section 3.6.2), `fovGoal` (section 3.6.2), `mapObstacles` (section 3.6.2), `mapDribble` (section 3.6.2), `mapKick2Goal` (section 3.6.2) and `mapReceiveBallFP` (section 3.6.2).

The last two presented maps have been vital for the development of new behaviors, namely the new dribble and kicking behaviors, as well as for selecting the ball reception position in free-play.

Figure 3.4 shows the situation created to illustrate all the different maps in the following sections. The team is attacking towards the yellow goal and Robot 4 has the ball engaged, hence being the Striker. Robot 3 is a MidFielder trying to get in a good position to receive the ball. Four obstacles have been placed in the field to simulate opponent robots.

Ball Field Of View Map

This map contains a Field Of View (FOV) map calculated from the ball position. The obstacles around the ball by less than 1.5 meters are ignored to avoid large zones with no FOV. The cells with no FOV are assigned the value of 1.0 and the others remain at 0.0. Figure 3.5 shows the height map calculated by Robot 4.

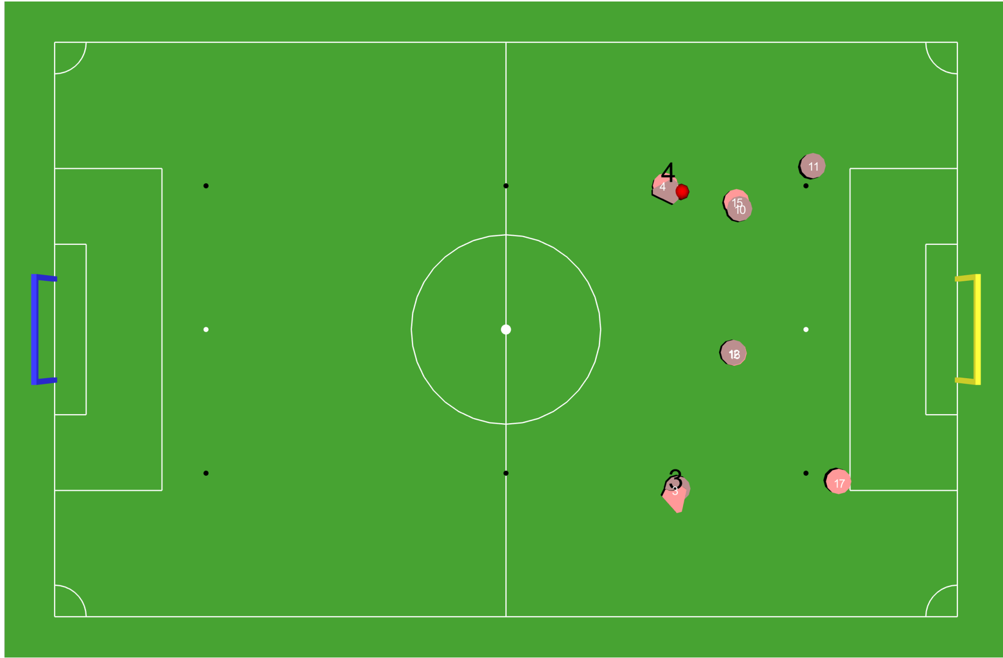


Figure 3.4: The example situation for illustration of the cost maps.

Goal Field Of View Map

This FOV map is calculated from the opponents goal. Cells with no FOV have a cost of 1.0 and the others have 0.0 cost. Because this map is used for evaluating the kicking conditions to the opponent goal. Since the kicks to the opponent goal are performed as lob-shots, above a certain distance to an opponent the ball passes over it, so cells with distances greater than 1.5m to an opponent aligned with the goal are also considered to have zero cost. Figure 3.6 shows the height map calculated by Robot 4.

Obstacles Map

The obstacles cost map (`mapObstacles`) includes hills in the position of the obstacles with 80% persistence to improve stability of decisions based on this map, which means that it takes 5 agent cycles (100 milliseconds) for a new obstacle to reach the maximum cost level. In the end, the map is normalized, thus holding values between 0.0 and 1.0. Figure 3.7 shows the height map calculated by Robot 4.

Dribble Map

The dribble cost map deals with the constraints regarding a generic dribble behavior, complying with the current MSL rules. Some areas of the field have high costs, namely both penalty areas, areas outside the field and outside a 3 meter radius circle centered on the point where the ball was grabbed. Then, the `mapObstacles` is added to the map and a cost associated with the robot movement is also considered. In Figure 3.8, the dribble map calculated by Robot 4 is depicted.

Kick2Goal Map

The `mapKick2Goal` represents the costs for kicking to the opponents goal. This map contains preferences such as the dribbling target position that can be easily changed in the existing parameter configuration tool. The `fovGoal` is added, because the robot needs to have free FOV for the opponent goal in order to perform a kick. Additionally, the `mapDribble` is also imported to comply with the rules while dribbling. Zones near the opponent corners also have raised costs, due to the small kicking angle to the goal. Finally, the map is normalized to produce floating-point values between 0 and 1. This map is used by the striker and is represented in Figure 3.9.

ReceiveBallFP Map

The `mapReceiveBallFP` is intended to select a good position on the field to receive a pass during free-play. Therefore, only positions inside the field are considered, as well as positions outside both penalty areas. Positions near the opponent corners (dead angles) are also avoided. A minimum distance of 2 meters to the Striker is required and preference is given to positions near the last chosen position, near the manually-defined strategic position and also near the actual receiver robot's position. Including the `fovBall` map makes the robot search positions with FOV for the ball. With these constraints, the free-play receiver robot will be constantly adapting to the changes of the opponent's formation and the ball position. Figure 3.10 shows the map calculated and used by the MidFielder (Robot 3).

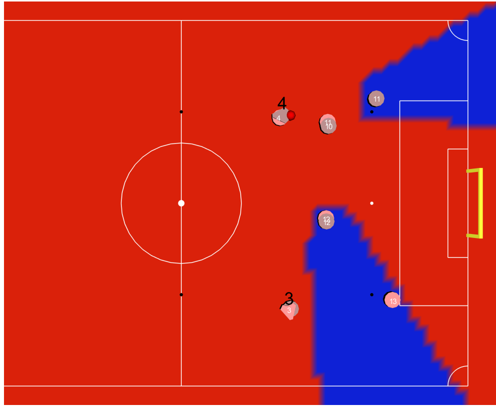


Figure 3.5: The Ball FOV map.

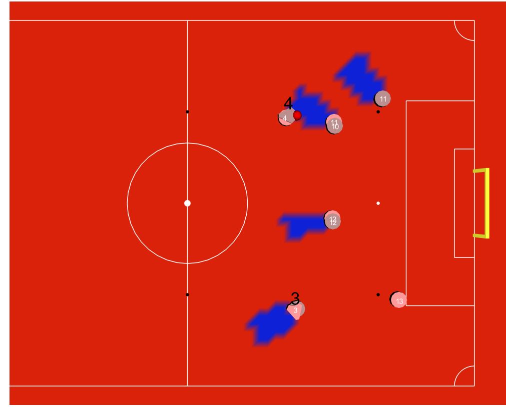


Figure 3.6: The Goal FOV map.

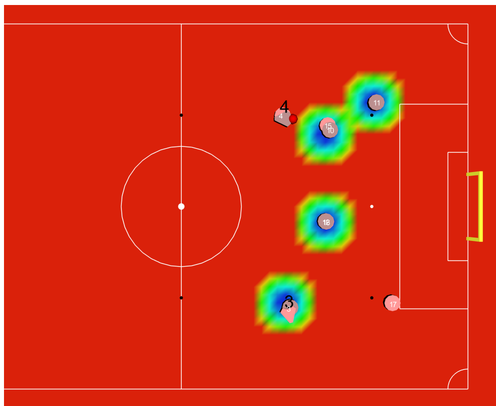


Figure 3.7: The Obstacles cost map.

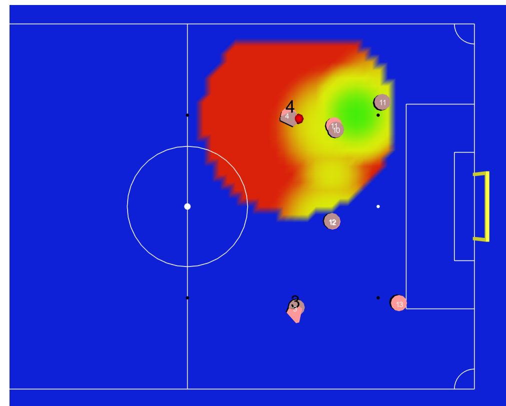


Figure 3.8: The Dribble cost map.

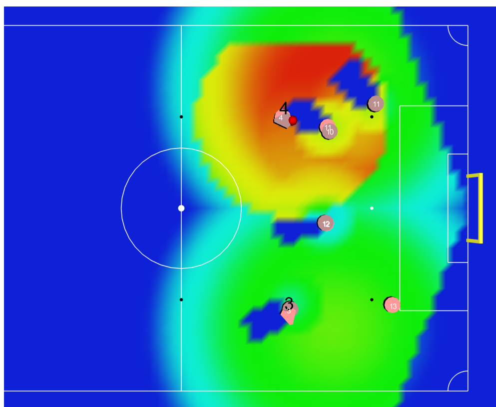


Figure 3.9: The Kick2Goal cost map.

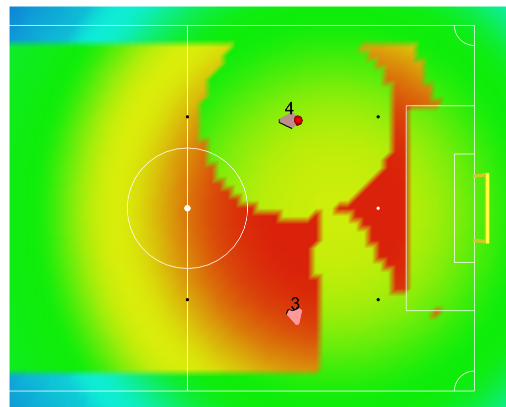


Figure 3.10: The ReceiveBallFP map.

3.7 Conclusion

In this Chapter, a new CAMBADA Software Agent Architecture has been described and proposed. The previous agent architecture has been successfully ported to this new solution, without any loss of functionality. After this software structure refactoring, making changes is now much easier than it was before. The `Behaviors` are well-defined and self-contained, and the Invocation and Commitment Conditions are directly associated to the behaviors themselves, which makes them a lot easier to tune.

If previously, when using state machines, all the transitions between states had to be hand-coded, now each behavior can ultimately be seen as a state, but the transition conditions are now implicit in the IC and CC of each behavior.

Furthermore, this structure makes it easier to develop new behaviors or to test specific ones. For example, to test the `BStrikerPass` behavior, all the other behaviors were removed from the arbitrator list, isolating that behavior for fine-tuning.

The new Software Agent architecture is modular and flexible, leaving space to use other types of compensators inside the `Controllers` other than the PID or other types of `Arbitrators`, such as utility arbitrators, which could select a behavior with the highest utility.

Because the `Roles` and `Behaviors` are instantiated once in the loading phase of the agent and the objects exists until the agent dies, the problem of losing history disappeared. Now both types of modules can easily maintain history, which can me wisely used to formulate new decisions.

The complexity of `Roles` has been decreased as shown in table 3.1 - line counting made at 4th of July, 2014. Although source lines of code is a very poor productivity measurement, having ported all the functionality and even added new behavious, the expressive numbers of the ratio prove that the requirement of relaxing the complexity of `Roles` has been met.

Role	Agent v1	Agent v2	Ratio
Striker	583	28	4.8%
Midfielder	193	29	15.0%
Goalie	176	28	15.9%
Replacer	541	94	17.4%
Receiver	675	322	47.7%

Table 3.1: Roles source lines of code comparison between architectures.

The developed Cost Maps have been very useful in the development of new behaviors,

namely the new dribble and kicking behaviors, as well as for selecting the ball reception position in free-play. The usage of height maps has broken some deterministic behavior and has improved the overall performance of the robots in such a dynamic environment as the MSL provides.

Chapter 4

New and Improved Behaviors

This Chapter includes all the behaviors developed in this thesis. The changes not only aim at adaptation to new rules for the Middle-Size League but also on the overall performance of the team, by tackling some of the weak points.

The first behavior to be studied and improved was the ball reception, since there was a noticeable low efficiency on that aspect by analyzing some of RoboCup 2013 matches videos. Then the 2014 rules forced a reimplementation of the dribbling behavior and coordination between robots to perform passes during free-play. On one hand, the changes made on the dribbling behavior increased the kicking chances, but on the other hand prejudiced the kicking efficiency, due to the robot's motion on the moment of the kick. Therefore, a closer look to the kicking behavior was taken. The system was characterized and a model was extracted in order to compensate for the robot's linear and angular velocities on the moment of a lob-shot.

4.1 Ball Reception Behavior - BReceiveBall

Receiving a ball in an efficient way is a very important aspect to achieve good results in soccer. Due to the high level of dynamism in the Middle-Size League games, the faster the receiver robot can engage the ball, the more opportunities to kick it will have.

After analyzing the games from last year, the ball reception behavior (`BCatchBall`) was one of the crucial aspects to be improved in the CAMBADA team, since most of the times the robot did not catch the ball at first. One of the last results [37] show a 10% efficiency of the `BCatchBall` behavior, which includes some calculations to be able to catch the ball with a certain velocity difference between the robot and the ball that is almost zero. To do that, it calculates the ball linear path from its velocity and when the robot is in the

trajectory line it starts to move back to catch the ball.

There are, however, two problems with this approach. Firstly, it relies on the ball velocity estimation, which is too erroneous to be used at this detail when the robot is moving fast. While moving in the field, the robot shakes, affecting the vision system pose and thus injecting noise in the perception of the robot position. Since the ball is firstly detected in relative coordinates (with some associated error) and then its absolute position is calculated based on the robot's position, both errors will sum, resulting in ball absolute positions that provide weak accuracy.

Secondly, when a long pass is made, the receiver may not see the ball in the first cycles and will use the information shared by the passer, namely, the ball position and velocity. While this may seem like a good procedure conceptually, it introduces some problems when we take it to a real-world situation.

The robot localizes himself on the field using the perceived field lines and the ball is detected in relative position, and then projected to absolute coordinates. The ball velocity is calculated using a linear regression of the absolute positions [38]. Of course, all of these measurements have errors, resulting in significant discrepancies between real and perceived positions. In addition to this, the information shared by the receiver has a considerable delay (approximately 100ms).

This abrupt change between shared information and local information can cause critical discontinuities that could lead to unusual behavior. On the other hand, discarding information that might be useful is probably not a smart decision. The challenge is how can we use information from both robots in an elegant and efficient way. In order to address the above stated problems, the `BReceiveBall` behavior was developed.

The first tests showed that with the new ball handling mechanism, the robot could engage the ball faster without moving backwards, so the developed work was mainly targeted at aligning with the ball trajectory.

Figure 4.1 may help visualize the algorithm behind this new behavior: when a robot (on the left) makes a pass, it calculates a **pass line** (dashed line) that is broadcasted to all the teammates. While the receiver (robot on the right) doesn't see the ball, he will move to the closest point on that line. When the ball becomes visible, a positional displacement of the ball in relation to that line is calculated (vector in red) and compensated from the closest point on the line.

In this approach, the trajectory of the ball is calculated by the passer robot, which is the robot that can potentially generate a better approximation, since it is closer to the ball from the first moment of the pass. The linear regression is performed for all the ball positions closer than a certain fixed (yet configurable) threshold.

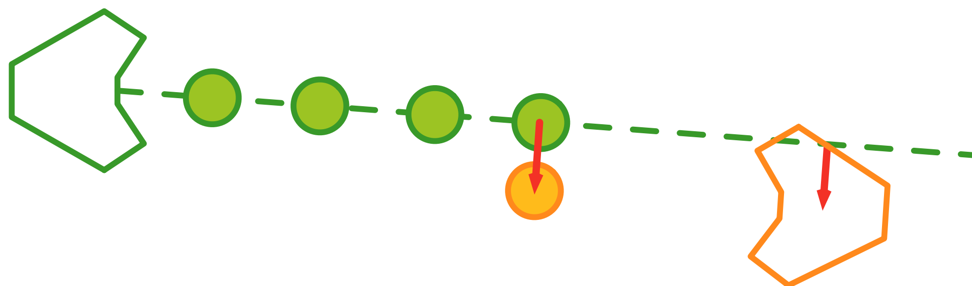


Figure 4.1: BReceiveBall graphical representation.

Obviously, if this robot has considerable positional error in its localization, this calculated pass line will also reflect that positional error. Nevertheless, **the slope of the pass line will not be affected** by this deviation and therefore, this slope is the most useful information. Although there is currently no way of calculating the robots' angular error, this can be considered negligible in this case.

The pass line is used to calculate a positional displacement to be corrected at each new cycle and, ultimately, the calculated position will converge to the best reception point, since it is calculated by the receiver robot.

4.1.1 Results

The efficiency of this behavior has been tested along with the ball passing algorithm. Therefore, the results are presented in Section 4.3.4.

4.2 Dribble - BBallBodyProtect

The changes in 2014 MSL Rules included a new limitation on the dribbling movement. Currently, a robot is not allowed to dribble the ball continuously more than 3 meters away from the point where it was grabbed. This means that to overcome this limitation, the robot needs to explicitly release the ball, passing to other robot or to grab it again. The whole dribble behavior had to be completely rethinked, since the movement is now much more limited and complex. Therefore, dribbling the ball can be one of these two situations:

- **Simply holding the possession of the ball**
  Example: I grabbed the ball, but still can not score a valid goal (I still have to make a pass), so dribbling means holding the ball while a teammate is trying to get in a good position to receive a pass.

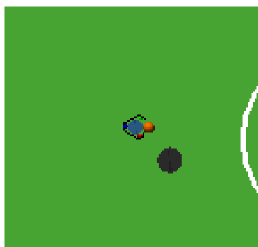
- **Dribble to a point where the ball can be kicked to the goal**
  Example: I have conditions to score a goal (by the rules), so now I have to dribble the ball to a point in the field that has free line-of-sight to the other team's goal while trying to avoid opponent robots.

Based on these requirements, a new behavior called `BBallBodyProtect` is proposed, which is based on a mixture of spacial and angular utility field maps.

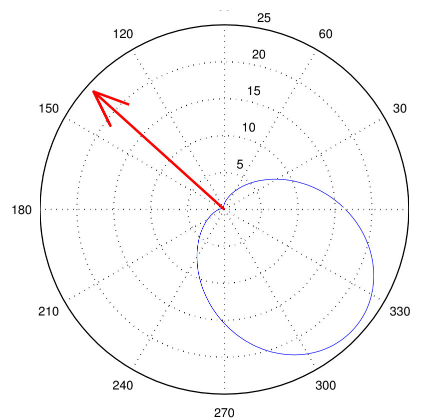
4.2.1 Implementation

The first step was to create a basic behavior that would make the robot turn his back to the opponents, while holding the ball, which is a very basic protective behavior.

One simple approach could be to iterate through each known obstacle and select the closest one to the robot and do the necessary calculations to turn the robot's back to that obstacle. However, a problem of **discontinuity** arises. If we do this, we can easily end up in a loop that keeps changing from obstacle to obstacle in consecutive cycles, which means unpredictable output and most certainly a loss of the ball's possession.



(a) Single obstacle near the robot.



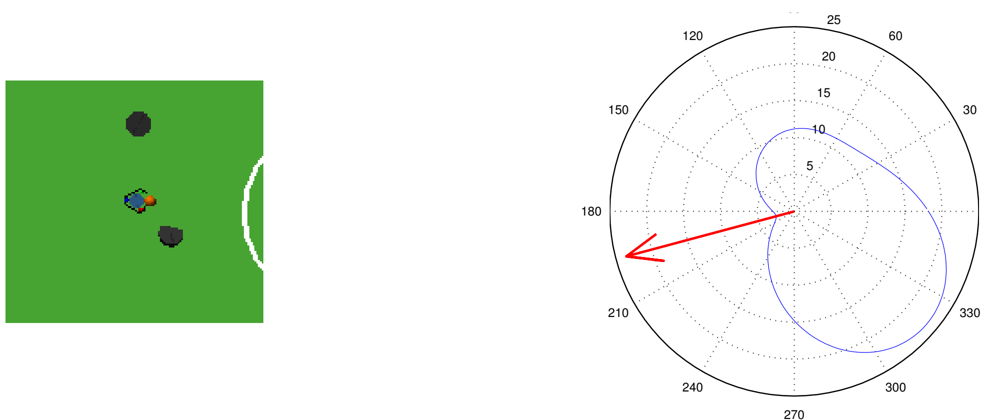
(b) Angular costmap and chosen relative angle (arrow)

Figure 4.2: Representation of the angular cost map for a single obstacle.

Therefore, an angular cost map was developed. Every obstacle within a certain threshold of relative distance counts to what this map is concerned, but they can have different weights from each other. A gaussian is added per obstacle, which will span a cost across other angles in its vicinity. In theory, if we chose the angle with the lowest cost, we are avoiding looking directly at the obstacle. Figure 4.2 represents the simplest situation with

just one obstacle. As Figure 4.2b shows, the chosen orientation angle is the opposite from the obstacle.

Now we consider the case when another robot from the opponent team approaches our `Striker`, like in Figure 4.3a, it is more or less intuitive to think that a closer obstacle should influence more the robot's behavior. In practical terms, this means that looking directly to the closest obstacle costs more than looking to the other obstacle. However, the goal is still to avoid looking at both.



(a) Two obstacles near the robot. (b) Angular cost map and chosen relative angle (arrow).

Figure 4.3: Representation of the angular cost map for two near obstacles.

So, in a situation like the one on Figure 4.3a, a gaussian is also added to the angular cost map, centered on the relative angle to each obstacle. Also, the magnitude of the gaussians decreases with the squared distance, so that closer obstacles have higher cost than distant obstacles. The result of the map can be represented on polar coordinates and can be seen in Figure 4.3b.

The cost of rotation is also taken into account, in order to improve stability. For instance, when there are no near obstacles, the robot keeps the same orientation and this can also help to choose between angles with similar costs - it makes the robot chose the closest one. This is done by adding values to the map that linearly increase with the module of the relative angle (*0.0* for relative angle *0º*, *K* for relative angles *-180º* and *+180º* , and linear distribution in-between).

Following the selection of the orientation, a target has to be chosen to position the robot, as it is counter-productive for a robot to stay always in the same place. The positioning in this behavior is done via the spacial cost map `mapKick2Goal` described in section 3.6.2. The behavior selects the position with the minimum cost value as the target position.

Having both the target position and orientation, the `cMove` controller is called to calculate the required velocities (X, Y and angular) to be sent to the low level and applied to the platform.

However, the controller was designed to make the robot move freely (and without the ball), so chances are that the output of the controller would make the robot inadvertently release the ball. To overcome that problem, we inject some more velocity in relative X, to account for the angular velocity, using the formula that relates angular and linear velocities:

$$v_\omega = \frac{v_l}{r}. \tag{4.1}$$

In equation 4.1, v_ω represents the angular velocity, v_l the linear velocity and r the radius of the movement. So if we use the distance between the robot center and the ball center when it is engaged, the result seems more like a rotation around the ball. This is why $v_\omega * r$ is added to v_x (linear velocity in relative X axis) after the controller call.

4.2.2 Results

The new dribbling constraints are too different from the previous to be able to directly compare both efficiencies. The most significant measurement in this case is probably the ball possession. In principle, a better dribbling algorithm will allow the team to have a higher ball possession ratio.

Therefore, the available log files from RoboCup 2013 have been compared with the ones from Robotica 2014 and an estimate of ball possession over the matches has been extracted. This estimate is not very accurate since it relies on the ball engagement flag reported by the robots, not taking into account situations when the ball is being disputed (engaged by both teams), thus not exactly in possession.

Figures 4.4 and 4.5 show the ball engagement ratio in RoboCup 2013 and Robotica 2014, respectively. The green dashed lines represent the average over the analyzed matches.

A small increase has been observed, with an average ball possession improvement from **5.9%** to **7.4%**. However, by visual inspection of the recorded videos of the matches in Robotica 2014, there were much less dispute situations. Therefore, the ball possession estimate of Robotica 2014 is much more accurate than RoboCup 2013.

By avoiding dispute situations, the CAMBADA team was able to create more attempts for passes or kicks to the opponent goal and the overall performance of the team has improved.

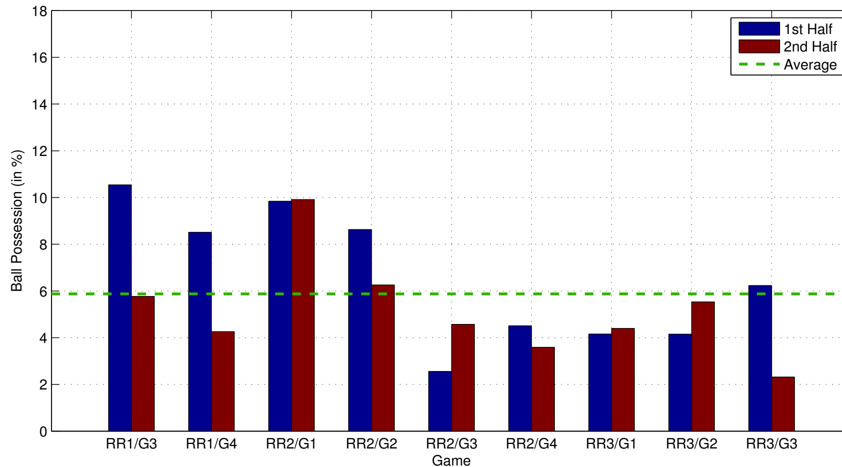


Figure 4.4: Ball Possession in RoboCup 2013.

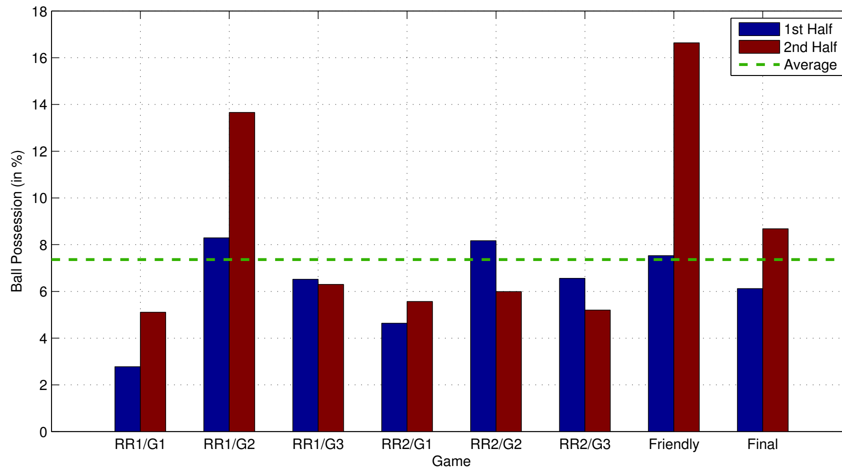


Figure 4.5: Ball Possession in Robotica 2014.

4.3 Free-Play Ball Passes

Passes between robots are already present in MSL for quite a few years, but most teams made them only during set-pieces situations (like Kick-Offs or Free-Kicks), others started to try out free-play passes. In 2009, some ideas have published [39] regarding the coordination of two robotic agents to perform passes in free-play. With the current league rules, it is mandatory for teams to be able to pass and receive a ball in free-play situations.

In the case of the CAMBADA team, passing is achieved using coordination variables - a **Coordination Flag** to signal that the ball is about to be passed, was passed or is in conditions to be passed, and a **Coordination Vec** (a position on the field).

In set-pieces like a Kick-off or a Free-kick, the team has 10 seconds to decide to make a pass, and can have a deliberative-like approach, with weak interference from the opponent team. Ultimately, a robot can always receive a pass in a set-piece if its position is between 2 and 3 meters away from the ball position, since the opponents are not allowed to be within a 3-meter radius around the ball in these situations.

A recent change in the rules states that the robots have to make a successful reception of the ball on the opponent side of the field in order to make a valid goal, which means that there has been a great effort in improving the free-play passing strategy. Being a true challenge in terms of coordination among robots, free-play passes are much more complex to achieve, due to the complete freedom the opponent team has to approach our Striker or cover a Receiver. Because the robot making the pass has the ball in his possession, its moves are very limited, so there was a huge effort to improve the receiver ability to select a great place to receive the ball. A possible solution is presented in sections 4.3.1 for the Receiver and 4.3.2 for the Passer.

4.3.1 Receiver

During free-play, one of the Midfielders is responsible for finding a good position to receive a pass. In terms of configuration, the possible receiver will be the Midfielder with the highest priority, since passes are so vital even if there are only two robots available.

In a first approach, the formation was manually defined with a position always on the opposite side of the field, possibly ready to receive a pass from a teammate. However, this is a static formation and it is now usual for the opponent team to cover our possible receivers, so a much more dynamic approach was needed. Then, the `mapReceiveBallFP` cost map (described in section 3.6.2 and illustrated in Figure 3.10) has been used for finding a good receiver position.

In order to synchronize the passer and the receiver, a coordination flag is communicated from the receiver to the passer: can either be *Clear* or *NotClear*. The `Striker` selects the free-play receiver by searching for the Midfielder with a *Clear* or *NotClear* coordination flag. *NotClear* means that the receiver is still moving to the reception position or is not in conditions to receive a ball (for example, there is no line clear between the two robots because an opponent is between them). On the other hand, *Clear* tells the Striker that this is a good opportunity to perform a pass. Additionally, a coordination vector is also shared to inform the passer of the receiver's target position.

4.3.2 Passer

In free-play the ball is passed from the Striker to a team-mate and therefore, passing is one of the possible behaviors for the Role Striker. But how can the agent decide between three possible behaviors (dribble, pass or kick)? The solution was to use the spacial height-map the Striker uses for positioning when dribbling the ball, but instead of just extracting the best position to be, the actual values of the map cells will be relevant to select the Behavior.

The height-map was used as a cost map, with normalized floating-point values between 0.0 and 1.0, so cells with lower values represent higher chances of keeping ball's possession, thus being the best cells to decide to kick. Then, the robot decides to kick to the opponent's goal when it is on a cell which cost is less than 0.25.

The passing behavior activates when the robot is aligned with the receiver within a certain angle, making the Striker align with the receiver and passing when the angular error is low enough. A higher difference on the receiver's relative angle releases the alignment with the receiver and the robot performs the fallback dribbling behavior (*BBallBodyProtect*).

4.3.3 Forward Passes

A forward pass consists in passing the ball to an area where there is no teammate nor an opponent, but ensuring that the teammate is still able to grab the ball. Making forward passes was a big necessity, since it is the solution to overcome the velocity of the adversary team. However, it is quite challenging to coordinate two robots to do intelligent passes, because it is too easy to release the ball and lose it on the pass.

A first approach on forward passes was made using some predictive components. The receiver's velocity was taken into account and used to predict its position in the future, assuming its module and angle will remain the same.

So, instead of just using the receiver's position, the Striker uses the receiver's coordination vector (the position where it is moving to) and velocity. The time of the pass depends on the pass distance, but to simplify it is assumed to last around 1 second. Consequently, the receiver's velocity (in m/s) is summed to its position and the result is used as an estimate of that robot's position in one second. If that position is near the target reception point, the Striker performs a pass to the latter. Figure 4.6 pretends to demonstrate the ideas behind the implemented forward passing algorithm.

Although this is a fairly basic approach, it performed well, since it did not seem increase the percentage of lost balls during passes, but there is certainly a big margin for

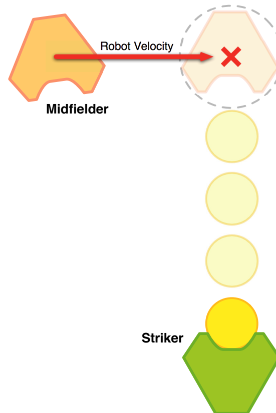


Figure 4.6: Forward Pass Algorithm.

improvement. Time constraints (time of Midfielder motion and time of ball travel) could be taken into account when calculating the best moment to perform the pass.

4.3.4 Results

To test the efficiency of the new pass (always forward passes) and reception algorithm, the RoleStriker Arbitrator has been changed to include only the dribble and pass behaviors. Two robots ran for 5 minutes, making consecutive passes while moving freely on the field, where 5 static obstacles have been randomly positioned. During the test, the robots have performed 63 passes, and the ball was correctly received 52 times, making a **82.5% success rate** for reception.

Some tests have been conducted also with dynamic obstacles. Although there are no actual measurements, the robots have struggled to perform a pass. However, they were very conservative regarding the release of the ball - it was passed only when there was a high confidence that the receiver would catch it. The ball has been passed to an opponent very rarely.

4.4 Kick to Goal - BKickToTheirGoal

After playing with the behaviors described in the previous sections of this chapter in IranOpen2014 (9th to 11th of April, 2014), CAMBADA has been able to create much more opportunities to kick to the opponents goal. Previously, all the kicks were made with the robot almost stopped, but this new behavior (`BKickToTheirGoal`) is much more dynamic and allows the robot to kick the ball while moving.

When the robot kicks the ball while having linear and/or angular velocity, the outcome will be different from the situation of when it is stopped. For example, if the robot is moving forward, part of the y-velocity of the robot (a factor representing the energy losses) will be added to the ball velocity in the kicking process. The same happens to the x-velocity and moreover, there will always be some delay between the instant of the kicking order and the actual kick of the ball. Therefore, the BKickTheirGoal behavior has been revised, as well as all the kicking calibration process.

4.4.1 Compensating the Linear Velocity

The linear velocity in the relative y-component was already being compensated, with the decrease of the kicking strength by a factor depending on this velocity. This affects the height of the kicking, but does not guarantee the correct alignment with the kicking target.

As previously stated, before the developed work in this thesis, the Kick behavior simply rotated the robot around the ball to align with a certain kicking target and as soon as the robot was aligned, it sent a kicking order to the low-level layer. Aiming at continuity between behaviors, the changes made on this behavior included having the robot selecting the same target position using the same method as the `BBallBodyProtect` behavior. The main difference is that now, it wants to orientate to a target laying on the opponents goal (may not be exactly the goal's center), instead of protecting the ball from the opponent.

As Figure 4.7 shows, the alignment with the chosen target (red cross) is evaluated based on the intersection of two line segments.

(a) Robot not aligned with the target.

(b) Robot aligned with the target.

Figure 4.7: Representation of the kicking alignment algorithm.

One of the line segments represents the robot's front direction and the other is centered on the target, perpendicular to the line between the robot and the target and with a certain length representing the allowed error for the intersection. The chosen error was *10cm* for each side of the target.

However, an efficient kick should send a kicking order even if the robot is not physically aligned with the target, but its motion (composition of instantaneous angular and linear velocities) would result in the ball being released to the target direction. This means that from the behavior's perspective, it needs to account for the robot's angular and x-axis velocities, and therefore each has been separately evaluated. In order to accomplish these objectives, two special behaviors have been developed to take measurements.

Figure 4.8 illustrates the test conducted to measure the influence of the robot's x-velocity on the kicking direction. The α angle was indirectly measured for several fixed x-velocities. The robot made a path parallel to the goal line, from one side line to another and kicks the ball as soon as it crosses the middle of the field. Then, the displacement to the center of the goal was measured to extract α.

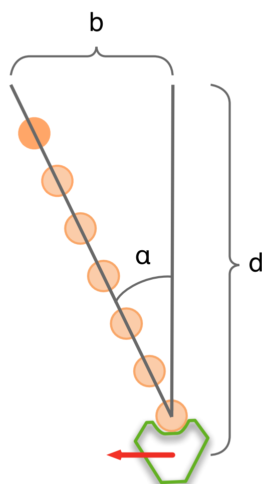


Figure 4.8: Influence of robot's x-velocity in kicking direction.

This was done for both positive and negative velocities with the conclusion that the outcome was symmetrical, and all the values are present in Table 4.1. More tests could not be made, because of the weak precision on the measurement of b (influencing the accuracy of α) and also for higher velocities, the robot is unable to hold the ball.

A linearization was done to come up with a function that outputs the α angle based on Vel_x. This angle is then used in runtime to rotate the robot's front line segment around the center of the robot and then it is used to calculate the intersection for the alignment.

Vel_x [m/s]	d [m]	b [m]	α [°]
0.5	6.0	0.5	4.76
1.0	6.0	1.0	9.46
2.0	6.0	1.8	16.70

Table 4.1: Measurements on the output angle for different robot's x-velocities.

4.4.2 Compensating the Angular Velocity

Finally, the influence of the angular velocity was evaluated and measured, using a special test behavior that makes the robot turn around its center with the ball at a certain fixed speed, as depicted on Figure 4.9. The robot was placed 8 meters away from the goal (d) aligned with the goal's center and the kicking order was sent as soon as there was an intersection between the two line segments. The allowed error (line segment above the target point, b) was manually adjusted in order for the ball to hit the center of the goal (the target, in this case). These two values allowed the extraction of the β.

The graph in Figure 4.10 shows a linear relation between the allowed angular error (in degrees) versus the robot's angular velocity (in degrees per second).

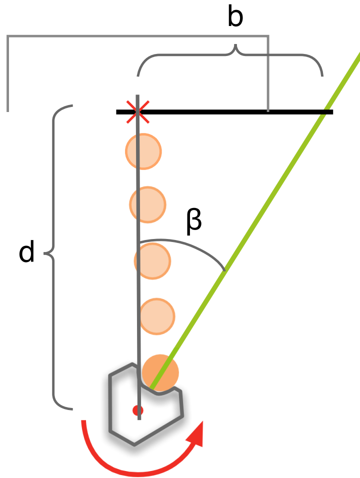


Figure 4.9: Influence of robot's angular velocity in kicking direction.

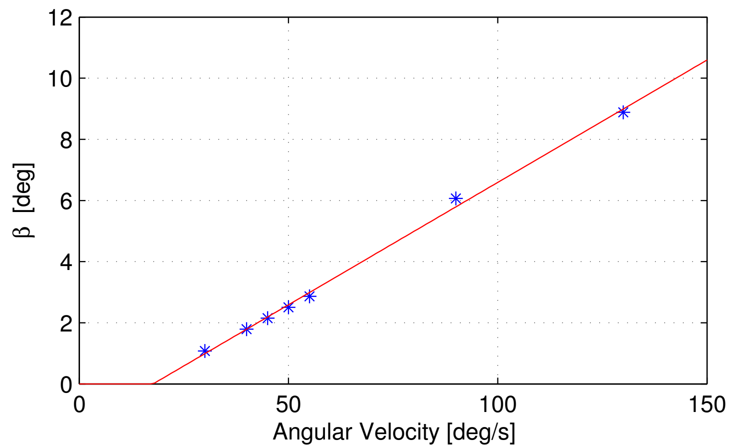


Figure 4.10: Allowed error vs Angular Velocity.

In runtime, the robot uses the odometry information to get a better approximation of the robot's angular velocity and then calculates the maximum β. The robot creates two line segments. The first line segment is created above the robot center position and

the kicking target and then it is rotated by α (explained in Section 4.4.1). The second is centered the target and is orthogonal to the line segment between the robot and the target, with *10 cm* to each side (the minimum allowed intersection error), by default. Then, using the β angle, it calculates the dimension of b, to grow the second line segment. If the robot is rotating with a positive angular velocity, the line will grow to the right and vice-versa.

As soon as an intersection between those line segments is found, a kicking order is triggered by calling the `DriveVector`'s `kick()` method.

4.4.3 Results

In IranOpen2014, before the described work was done, the kick efficiency was as Table 4.2 shows. An average of **40,6%** of kicks would never translate into valid direct goals, since the ball was kicked outside of the field or to a post or a bar.

Phase	Opponent Team	Inside Goal	Outside Goal	Efficiency
Friendly Matches	MRL	5 (3 goals)	0	100.0%
	Hibikino	4 (3 goals)	4	50.0%
	MRL	2 (0 goals)	4	33.3%
Round-Robin 1	Hibikino	6 (3 goals)	5	54.6%
	MRL	1 (0 goals)	0	100.0%
Round-Robin 2	Hibikino	8 (5 goals)	6	57.1%
	MRL	6 (2 goals)	4	60.0%
Final	MRL	2 (2 goal)	8	20.0%

Table 4.2: Kick accuracy during IranOpen2014.

Table 4.3 shows the efficiency of the developed method in the Robotica2014 competition. In average, **74.0%** kicks were in goal, proving that the developed work resulted in an efficiency improvement of **14.6%**.

4.5 Contouring Obstacles - BStrikerContourObstacle

During IranOpen2014, the CAMBADA team has made quite a few foul tackles by hitting an opponent robot that has the ball, making it lose the ball's possession. Most of the times, the CAMBADA robot's intention was to go to the ball, but without any consideration of the opponent robot.

Phase	Opponent Team	Inside Goal	Outside Goal	Efficiency
Friendly Matches	Tech United	3 (1 goal)	1	75.0%
	Tech United	5 (0 goals)	2	71.4%
	Carpe-Noctem	17 (10 goals)	3	85.0%
Round-Robin 1	Tech United	1 (1 goal)	0	100.0%
	Carpe-Noctem	15 (6 goals)	6	71.4%
	5DPO	17 (11 goals)	4	81.0%
Round-Robin 2	Tech United	1 (1 goal)	1	50.0%
	Carpe-Noctem	12 (8 goals)	5	70.5%
	5DPO	17 (12 goals)	1	94.4%
Semi-final	Carpe-Noctem	7 (4 goals)	3	70.0%
Final	Tech United	5 (1 goal)	6	45.5%

Table 4.3: Kick accuracy during Robotica2014

Therefore, the behavior responsible for going to and grabbing the ball was had to be reviewed to take into account a possible obstacle that might be on the way. The behavior `BStrikerContourObstacle` was then developed, which objective is to contour an adversary with the ball until they are both almost face-to-face to be possible to directly attack the ball.

4.5.1 Implementation

The developed and proposed algorithm is based on the usage of an intermediate waypoint between the current position of the robot and the target (the ball), in a way that it will keep a certain minimum distance to the opponent holding the ball. The behavior was developed to do this while protecting our goal, so the applied algorithm varies with three different situations.

In the situation where the **ball is visible**, an **obstacle is detected dribbling it** and the **ball is closer to our goal than the robot**, the waypoint is calculated using the intersection between two circles. One circle is built around the ball with a fixed radius and the other has its center on the robot and the radius is equal to its distance to the ball. Then, in the first cycle, the intersection that is closer to our goal is picked and in the consequent cycles, the intersection is picked based on minimum distance to last cycle's target. Figure 4.11 visually demonstrates this algorithm, with the opponent in black, the robot in green and the green cross representing the selected intersection.

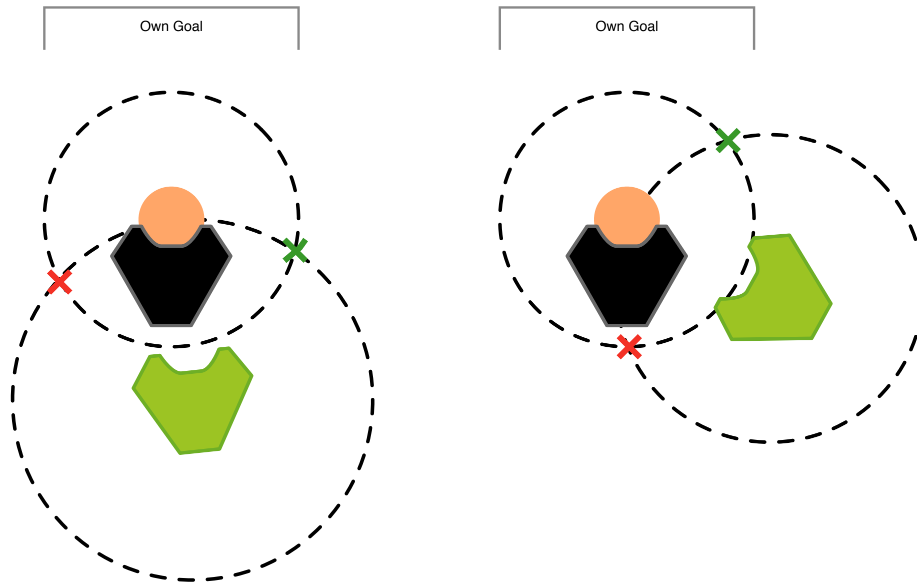


Figure 4.11: BStrikerContourObstacle contour algorithm.

When the **ball is visible**, an **obstacle is detected dribbling it** and the **robot is closer to own goal than the ball**, a protective behavior is implemented and the robot positions himself between the goal and the ball (Figure 4.12).

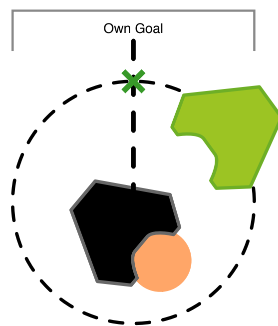


Figure 4.12: BStrikerContourObstacle protective behavior.

A wise decision for the opponent would probably be to protect the ball from our robot. It might turn around and the ball's position might become unknown and therefore invalidating the developed algorithm. This is why this behavior is slightly different when the **ball becomes not visible**. In this situation the behavior tries to track an obstacle from cycle to cycle and the same protective algorithm is applied to the tracked obstacle instead of the ball, as demonstrated on Figure 4.13.

This tracking method relies on using the last known position of the ball in the first

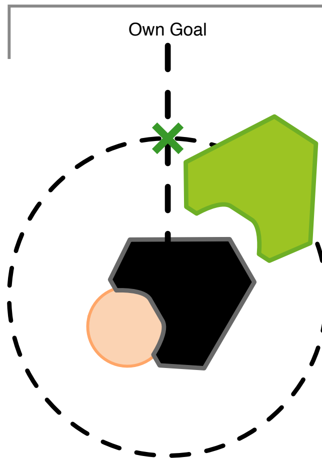


Figure 4.13: BStrikerContourObstacle protective behavior without knowing ball position.

---

cycle it becomes invisible. After that first cycle, all the obstacles near the last tracked position are aggregated into a cluster and the mean position of the cluster is used as the tracked opponent. This is required because sometimes during fast movements of the robot or the opponent or due to improper calibration, some perception errors can occur, resulting in seeing multiple obstacles in a single real opponent. Using the mean of these perceived obstacles is much more accurate than simply using the closest obstacle of the last tracked position.

The Commitment Condition becomes false when both robots are almost face-to-face, letting the parent arbitrator call a direct ball approach behavior (*BGoToVisibleBall*).

4.5.2 Results

This behavior has been included in the `RoleStriker Arbitrator` list for Robotica2014. In the final match of the tournament, the `BStrikerContourObstacle` behavior has been selected by the `Arbitrator` **34%** of the time in the first half and **37%** on the second half, having enhanced the defensive attitude of the team, by blocking many of the opponent's dribble attempts towards our goal.

Chapter 5

Debug and Development Tools

In order to develop the new architecture and test the new behaviors, there was the need of improving some of the current development tools or to create new ones for visualization and calibration. The Basestation application [14] has been improved to allow a 3D visualization with support for a height map visual representation. To test some behaviors that are dependent on the opponents' dynamic behavior, the team needed a tool that could simulate a dynamic opponent in realtime, so an Augmented Reality tool was developed for this purpose. Lastly, the deep study of the kicker system led to the creation of two calibration tools during the process. Those are also presented in this chapter.

5.1 Basestation 3D Field View

The Basestation (Figure 5.1) is a crucial application used both in the competitions and in the development phase to start and stop robots, assign roles, debug behaviors, simulate referee-box signals, etc. This QT4 application contains buttons and other widgets that can be used to interact with the robots and includes a visualization of the worldstate representation.

The objective of the work mentioned in this section was to replace the current 2D Visualization in the Basestation by a 3D view, maintaining the existing additional information regarding each player and for specific game situations.

The Basestation application already has a module responsible to store all the information transmitted by the robots and provide it to the various Widgets, so the developed work was basically the replacement of the `FieldWidget` (the main visualization part). This widget was previously done in low-level OpenGL calls, which require some time to be able to modify. Therefore, the Visualization ToolKit (VTK) was chosen as a framework for this

Figure 5.1: The 2D Basestation

---

visualization widget, because it provides an intuitive way of manipulate the 3D objects on the scene with a high level of abstraction from the low-level OpenGL calls.

Before starting the development of a new solution, the existing one was studied at detail, and a list of features were noted, in order to keep full functionality over the transition to the new solution:

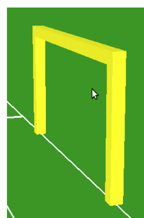
- **Permanent Actors**
  - **Field lines** - the field dimensions are read from the XML configuration file and are drawn accordingly.
  - **6 Robots** - the old robots were round and the new ones have flat surfaces
  - **6 Balls** - each robot sees the ball in different positions and all of them have to be represented in the field.
  - **Ball Velocity** - the ball velocity estimation was drawn based on the information from the closest robot to it.
  - **Debug Points (4 per robot)** - there are 4 available coordinates that are shared by the robots. Those are used for debug purposes and should be visible from the basestation view.
  - **Score Board** - this was not present in the current 2D visualization, but makes sense to include it in order to become a full functional visualization widget that can be used out of the basestation - for instance, to display some information to the audience.

- **Variable Actors**
  - **Obstacles** - each robot can see multiple obstacles around him, but will only share up to 6 of those obstacles. Of course, in some situations, the robot will see less than 6, can even happen that he does not see any obstacle. Therefore, the number of obstacles to be drawn are arbitrary.
  - **Set-pieces lines** - for some specific game situations such as set-pieces (for instance, kick-off, corner-kick, throw-in, etc), some debug lines are drawn based on some of the information reported by the robots: receive position, pass line clear between ball and receiver position.

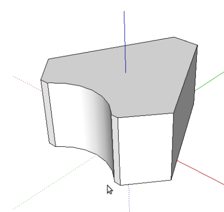
To carefully handle the memory, a dynamic array is available to store the variable actors. Then, at the next update cycle, in the beginning, all actors in that array are deleted from the scene and their associated memory is freed.

5.1.1 3D Models

Two (***.obj**) 3D models were created: **goal** (Figure 5.2a) and **cambada_base** (Figure 5.2b), with the help of Trimble Sketchup [1]. The models were kept as simple as possible in order to maintain a good performance in slower computers. All 3D models are loaded and the actors are added to the scene in the loading phase.



(a) The model for the yellow goal.



(b) The model for the robots.

Figure 5.2: 3D Models created for the new Basestation Field Widget.

5.1.2 Colors

In the 2D Basestation, there were 6 different colors assigned to each robot. Therefore, the robot body, the ball and the obstacles seen by that robot will all have the same color. These were maintained because sometimes it is useful to see which is the robot seeing a ball (sometimes it is a false-positive that will require extra vision calibration in that robot).

[1] http://www.sketchup.com/

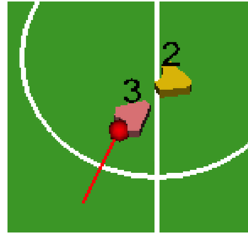


Figure 5.3: Robot 3 has the ball engaged.

When a robot has the ball engaged, there was a stroke drawn around the circle that represented the ball in the previous 2D view. In VTK, it is very difficult to achieve a similar result, therefore there was a change in the way the ball is represented when it is engaged: now, its size is bigger than a regular ball and its color is changed to red (Figure 5.3).

5.1.3 Set-pieces

For the Set-pieces (kick-off, corner-kick, throw-in, etc), some dashed lines are drawn based on the information reported by the robots. As Figure 5.4 shows, there is a dashed-line drawn between the passer and the point where the receiver will receive (may not be exactly his current position), which is green when there is a free line of passing and changes to red if an obstacle prevents a successful pass.

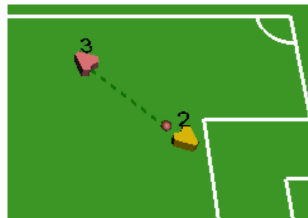


Figure 5.4: A Free-Kick set-piece example.

5.1.4 Widget Integration

After developing the basic functionality of the `FieldWidget3D`, it has been integrated in the Basestation application (Figure 5.5). Because this is such a critical application and in order to demonstrate the robustness of the solution, the new Basestation was intensivelly used for several days and no problems occurred.

Figure 5.5: The 3D Field Widget integrated into the Basestation.

5.1.5 Interaction

In VTK terminology, the Interactor is the module that defines a way of manipulating objects or the camera on the scene. In this case, it is basically a mouse interaction. `vtkInteractorStyleTerrain` was chosen as the base `Interactor` style, since it enables the user to manipulate the camera in scene with the natural view up.

Using the left mouse button, the user is able to manipulate the camera's azimuth (angle around the view up vector) and elevation (the angle from the horizon). Using the middle mouse button, the user can pan the camera around the scene. Finally, it is possible to zoom in and out by holding the right mouse button and moving the mouse up and down.

Furthermore, the base `vtkInteractorStyleTerrain` has been extended to empower the Basestation functionality. A different interaction is activated when the user clicks on a robot. In these situations, the user is able to drag a model of a robot to a desired position. This will trigger a new `Role` in the respective agent, called `RoleTaxi`, developed to relocate a robot in the field without physical interaction.

5.1.6 Height Map Visualization

Since the development of the CAMBADA agent is evolving in a way that uses more and more Height Maps, there was a need of adding a visualization of the `Height Map` to the `FieldWidget3D` (Figure 5.6. This mode can be activated in the Field menu and can

be shown as a color map or as a grey gradient and can also be displayed as a flat plane or in 3D.

This mode has been extensively used when developing new behaviors that rely on height maps for positioning and decision.

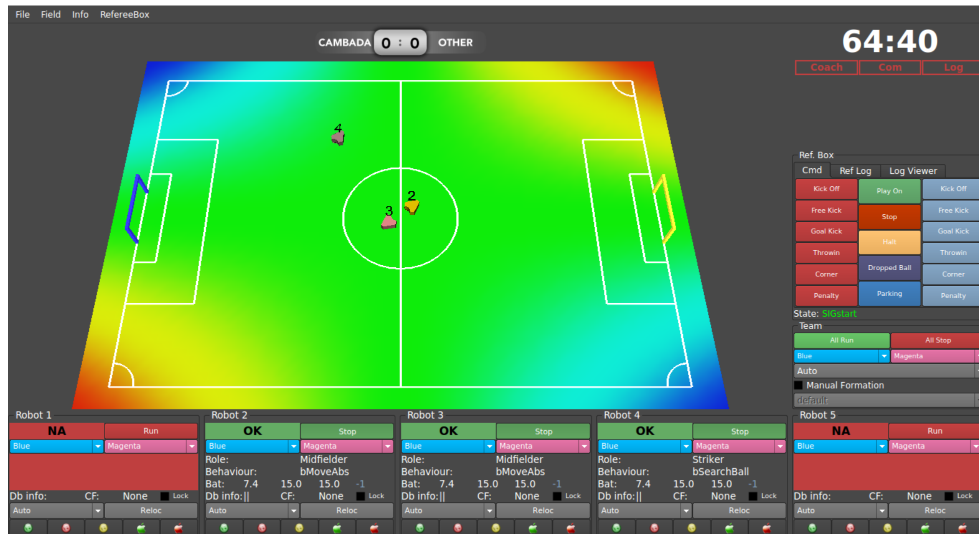


Figure 5.6: Colored Height Map in flat mode.

5.1.7 Results and Future Work

In terms of the objectives for this particular change, all of them were fulfilled: a new 3D visualization for the basestation was successfully developed and integrated in the application. Additionally, more information is displayed to help in the debugging process. VTK was particularly useful for visualizing data such as the virtual potential fields directly on the Basestation, which would be very difficult to implement using basic OpenGL functions. Moreover, it is now possible to represent the ball in the 3D space, instead of just on the field plane.

The developed widget can easily be integrated in a different application for representation purposes, for instance, for the audience during games.

There are some things that could complement the developed work, namely:

- A dynamic active camera that follows the ball (for demonstration purposes).
- Measurements over the performance dropage with high-detailed models of the robots.

5.2 Augmented Reality Visualization

Virtual Reality (VR) and Augmented Reality (AR) are emerging research areas, with limitless applications. The developed work included a first approach on bringing AR into the CAMBADA environment and using it as a tool to support development. Furthermore, due to the engaging experience and the general interest over this technology, the developed work could possibly be also used for demonstration purposes.

Markers are features or objects that exist in the real world that, when they are within the Field of View of the camera, allow the AR application to use them as a reference guide that enables to join the virtual and the physical world. Although Augmented Reality is possible to achieve without markers (called Markerless AR), in this application bitonal markers have been used, since they provide a much more accurate estimation of the camera's pose (position and orientation).

In order to achieve the results, two external libraries were used: **ARToolkit** and **VTK**. The first already has methods to access video and image capture devices, and tools for detecting markers, giving direct access to translation and rotation matrices and some transformation functions. VTK was picked for display purposes, since it was possible to use a lot of the work previously developed for the Basestation 3D view (section 5.1).

In a first stage, two layers (Figure 5.7) have been configured in a separate VTK window. The bottom layer is responsible for representing the captured image from the camera, simply by having a `vtkImageActor` without perspective, covering the entire window. The upper layer has a 3D scene `vtkRenderer`, in which the virtual world is represented, above the captured image.

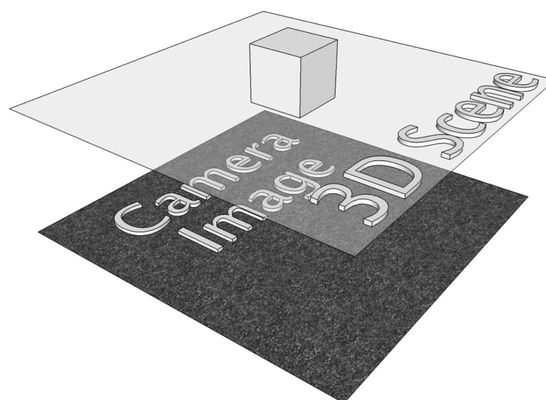


Figure 5.7: The two layers in VTK for the AR application.

In this type of applications there are typically two options to what the representation of the virtual world concerns:

1. **Origin in the camera** - the camera is fixed at the origin of the world and the translation and rotation transformations are applied to all the objects.

2. **Origin in the marker** - we position all the objects in the virtual world in their natural position and move the camera around, replicating the relative movement of the real camera to the marker.

Given the objectives of the work and what was already done, the second approach was chosen, establishing the origin of the world on the marker position (Figure 5.8), thus reducing in a large-scale the number of operations (because it drops the need of applying transformations to all objects per frame).

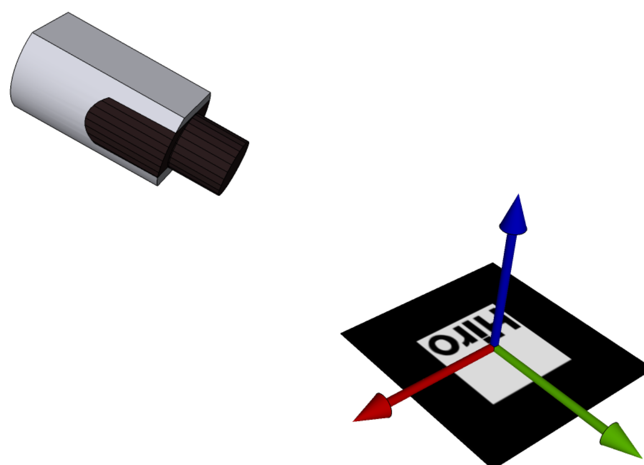


Figure 5.8: World axis centered in the pattern.

In VTK, the pose of the camera is set by calling three methods - `SetPosition`, `SetViewUp` and `SetFocalPoint`, each one requiring a three-dimensional vector that are either directly given by ARToolkit or can be calculated using matrices returned by that library.

The camera clipping range has also been extended to allow the representation of objects in shorter and longer distances. In this particular case, the only limitation is the distance range at which the marker can be detected. Finally, the camera's Field of View was also adjusted to match the camera that was used.

5.2.1 Marker Detection

The detection of the markers is made by ARToolkit (ATK), which has its own coordinate systems [2] (represented in Figure 5.9). Therefore, in order to get the correct values for the expected result, some operations need to be performed over the matrices returned by ATK.

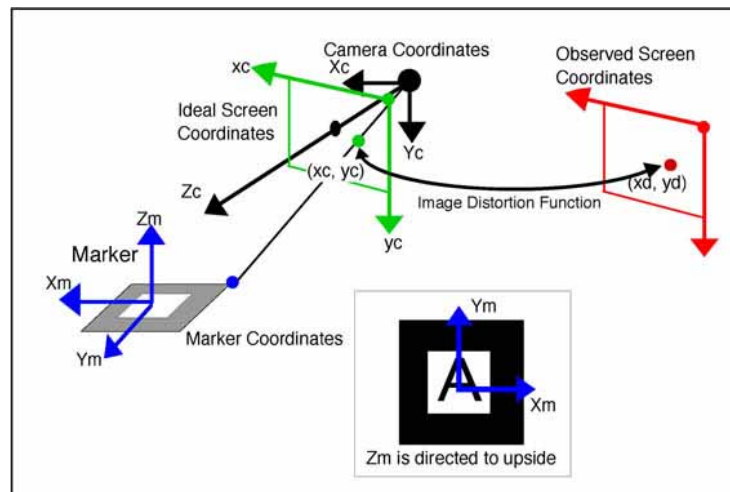


Figure 5.9: Coordinate systems used in ARToolkit.

The `arGetTransMat` method from ARToolkit returns a matrix M, which includes the position the marker in the camera coordinates. Its inverse (M^{-1}) is then calculated using the `arUtilMatInv` method. The second column of M^{-1} represents the **view-up** vector directly, but to synchronize it with the VTK coordinate system, all the components had its signal inverted.

It is also possible to calculate the homogeneous transformation matrix for OpenGL (column-major) H using the `arglCameraView` method, that treats the marker as lying in the x-y plane, with the +z axis pointing towards the observer. This matrix contains the **position of the camera** $(x, y, z) \rightarrow (H_{12}, H_{13}, H_{14})$. The **focal point** vector can be obtained by adding (H_8, H_9, H_{10}) to the position of the camera.

At this point, all the three needed vectors (view-up, position and focal point) to replicate the behavior of the real camera in the virtual VTK scenario are known.

[2] `http://www.hitl.washington.edu/artoolkit/documentation/cs.htm`

5.2.2 Interaction for Development Support

Currently, when developing new behaviors that require tests with an opponent robot, there are only two possibilities:

1. Using the simulator, placing one or more static obstacles in the field, and performing tests with a static opponent team.
2. Take the test in real robots, placing a turned off robot on the field, which can be moved by one person, thereby recreating an opponent behavior.

However, neither of these solutions is practical. The second option requires at least two people (one to move the obstacle and another to monitor the behavior of the robot and stop it if necessary). The first option is currently the most used, but has the disadvantage of conducting tests with a static opponent team, which never corresponds to reality.

Therefore, in an attempt to reach a viable and more practical solution to test and refine behaviors that are dependent on the opponent dynamic behavior, a solution was thought: in our AR scenario, we can rotate the camera in any direction, and know its exact position and orientation (as long as the marker stays visible in the image), one can use the camera as a pointer, which target is the point in the field intersected by the camera's optical center (Figure 5.10).

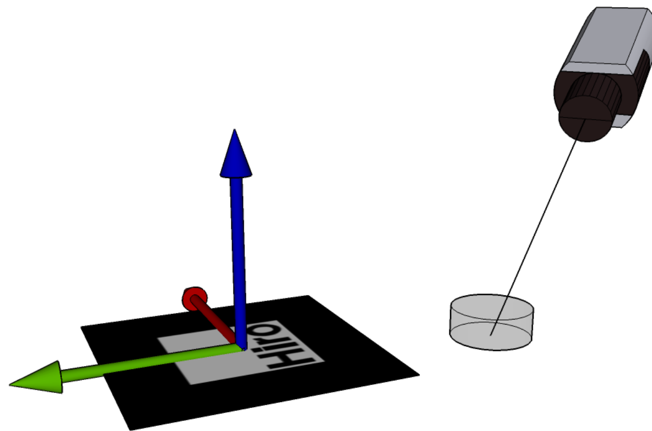


Figure 5.10: Intersection of the optical center line with the x-y plane as the target.

The objective is to use that intersection point to inject a virtual obstacle to the agents list of obstacles. By maneuvering the camera, we keep updating the virtual obstacle posi-

tion on the field, making it very easy for the user to simulate the behavior of an opponent robot and test the behavior, both in a real CAMBADA robot and in a simulated environment, because the RtDB creates this level of abstraction. The information regarding the virtual obstacle is stored in a shared item, which is broadcasted to the real robots by the **comm** process or to the simulated agents by the simulator itself.

To achieve this, the Integrator module in the agent (responsible for integrating all the information gathered from the various sensors and also the information shared by the teammates) had to be changed. One of the tasks of the integrator is to merge all the obstacles shared by the teammates and the ones seen by the robot itself into a single list of obstacles. At this point, we introduced a fake obstacle in the list, in the position shared by the AR application.

5.2.3 Results

As a proof of concept, the "hiro" pattern was printed in a A4 paper sheet and was used as marker for the application. Although any webcam would work, Microsoft Kinect was chosen, because it has decent resolution and is physically easy to manipulate with both hands. The setup is represented in Figure 5.11.

Figure 5.11: The setup used to test the application.

The final result can be seen in Figure 5.12, where the five robots are represented in a

Kick-off situation, and in the center of the image (the projection of the optical center), the virtual obstacle is represented as a black cylinder. When the user moves the camera, the virtual obstacle also moves on the field.

This type of interaction enables us to interact with the robots (whether they are real robots or in a simulation environment) to test behaviors that are dependent on an opponent's dynamism.

AR was introduced for the first time in the CAMBADA architecture. A new module and application was developed to explore this area and conduct the first experiments, meeting all the requirements set for both possible applications: the demonstration for audience in the robotics venues and also a new way of interacting with the robots, by using a camera as a pointer to the virtual world.

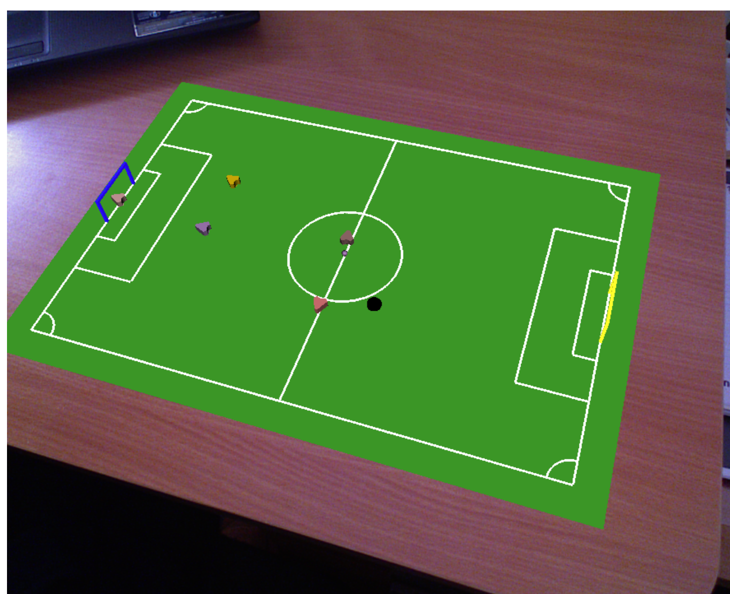


Figure 5.12: Final result of the AR application.

5.2.4 Future Work

The developed application is a good starting point to explore the AR world in the CAMBADA environment and therefore was integrated into the repository to be used and extended by all the team members. Although all the objectives for this application were achieved, there are some possible improvements, which are presented in this section.

Using a paper sheet with just one marker printed in the center limits the camera orientation range, because the marker needs to stay visible in the image to have an estimate

on the camera position. To overcome this problem, a paper sheet filled with markers could be used, which would not only extend the range of the camera freedom but also give a better estimate on the camera pose in case there is more than one marker visible.

For each new frame, a camera pose is estimated and applied to VTK directly. Due to noise in the acquisition, a small jitter can appear in the estimation and that can turn down the AR experience. ARToolkit provides a method that uses the last frame detection in order to improve stability that could be used. Adding a filter to the camera position and orientation may also be a good improvement.

A little fine-tuning is needed to fully synchronize the two scenes (the captured frame and the 3D virtual world), since there is still a little positional offset between them. Maybe there are some parameters of the *vtkCamera* that can be explored to have them perfectly matched.

Instead of having to look at the monitor to see the virtual world, it would be great to use the camera as a pointer in the real field. This represents a big challenge, since there is not much interest in having patterns inside the field, but some could be placed outside the field, to serve as landmarks, and by attaching an IMU to the camera, its pose could be tracked when there are no visible landmarks.

A bigger step would be to drop off ARToolkit and, using just OpenCV functions to extract interest features, estimate the relative position of the camera on the real field. The suggestion regarding the IMU also applies here.

An alternative to having a camera serving as the pointing device, is to have a camera outside the field and a marker serving as the pointer. Again, having a sensor attached to the marker would be beneficial.

Chapter 6

Conclusion and Future Work

This main objective of this thesis was to improve the structure of the agent, from the code organization to the actual software architecture, while streamlining the whole process of developing, testing, debugging and refining agent behaviors. An introduction on Robotic Agent Architectures was done and different solutions from other teams have been presented and the weaknesses of the CAMBADA agent architecture were identified, namely **lack of history** in `Roles` and `Behaviors` and **high complexity** of the `Roles`. Eliminating or relaxing both problems was a requirement.

6.1 Conclusion

At the time this thesis started, the CAMBADA agent architecture was behavior-based, with Finite State Machines with hand-coded conditions for decision and some real-time adaptive dynamism. The base strategy was defined beforehand using calibration tools, but minor adjustments were made in gameplay based on the opponent's team attitude.

Based on the above-stated requirements, a new solution for the CAMBADA Agent Architecture has been presented, with the introduction of new modules responsible for producing a `DriveVector` (a new struct that holds the low-level information that is sent to the platform). A unique `DriveVector` object currently exists and is completely orthogonal to the various decision layers, in order to extend the flexibility of the architecture.

The `Controllers` have been introduced as the modules responsible for calculating the desired robot velocities, based on a certain target(s), which may vary from `Controller` to `Controller`. Four different controllers (`CMove`, `CArc`, `CRotateAroundTheBall` and `CRotate`) have been implemented to be able to achieve different robot movements in distinct situations.

Furthermore, the Behavior interface has been extended to include four additional methods: checkIC that returns the Invocation Condition, checkCC that returns the Commitment Condition, gainControl and loseControl that are two callbacks that can be used by the Behavior, for instance, to initialize variables and/or to free allocated memory when needed.

As an alternative to FSM, the CAMBADA Arbitrator has been developed to select and execute a certain Behavior from a list of options, based on their Invocation and Commitment Conditions. The developed Arbitrator has the possibility of mixing the ideas of **Priority** and **Finish Plan First** arbitrator types, since any Behavior can be marked as **non-interruptible** when being assigned to the Arbitrator options list.

If previously, when using state machines, all the transitions between states had to be hand-coded, now each behavior can ultimately be seen as a state, but the transition conditions are now implicit in the IC and CC of each behavior.

Roles now have an Arbitrator by default, as well as a callback method that can be implemented by the derived Roles to make decisions and/or update internal variables or own worldstate information. In the constructor of the Role, a set of behaviors is added to its Arbitrator queue.

Because the Roles and Behaviors are instantiated once in the loading phase of the agent and the objects exist until the agent dies, the problem of losing history disappeared. Now both types of modules can easily maintain history, which can me wisely used to formulate new decisions.

Moreover, the developed structure makes it easier to develop new behaviors or to test specific ones, by isolating that Behavior on the Arbitrator for fine-tuning. The new Software Agent architecture is modular and flexible, leaving space to use other types of controllers inside the Controllers other than the PID or other types of Arbitrators, such as utility arbitrators, which could select the Behavior on the list with the highest utility.

The complexity of Roles has been relaxed, resulting on 80% reduction of source lines of code regarding the previous version. On the other hand, the number of Behaviors has increased, as well as their respective lines of code. However, the behaviors are well-defined and self-contained, and the Invocation and Commitment Conditions are directly associated to the behaviors themselves, which makes them easier to tune.

The HeightMap class enabled the development of a series of useful cost maps for new behaviors, namely the new dribble (BBallBodyProtect) and kicking (BKickToTheirGoal) behaviors, as well as for selecting the ball reception position in free-play. The usage of height maps has broken some deterministic behavior and has improved the overall perfor-

mance of the robots in such a dynamic environment as the MSL provides.

Some new behaviors have been developed in order to comply with the new rules and to keep the high competitiveness level of the team.

Regarding the new dribble behavior (`BBallBodyProtect`), a small increase on ball possession in matches has been observed and, by visual inspection, there were much less dispute situations. By avoiding these dispute situations, the CAMBADA team was able to create more attempts for passes between teammates or kicks to the opponent goal, while complying with the rules defined for 2014.

The 2014 MSL Rulebook also foments the passes between robots and therefore, the reception and passing algorithms have been revised. The `BReceiveBall` has been changed to optimize the reception behavior for the new platform, making use of the shared pass line from the passer robot, but adjusting to overcome any inconsistency. Also, making passes during gameplay is a true challenge in terms of coordination among robots, due to the complete freedom the opponent team has to approach our `Striker` or cover a possible receiver. There was a huge effort to improve the receiver intelligence, using a spacial cost map to find the best reception position on the field. Some tests conducted on the laboratory showed a **82.5% success rate** for the passing (including some forward passes) and reception behaviors.

With the developed work on the `BKickToTheirGoal` behavior, an efficiency improvement of **14.6%** has been observed regarding the accuracy on goal, resulting in more chances of scoring goals, which is one of the most important objectives in soccer.

Additionally, a protective and defensive behavior (`BStrikerContourObstacles`) has been developed for Robotica2014, having enhanced the defensive attitude of the team, by blocking many of the opponent's dribble attempts towards our goal, while foul plays.

Also in the context of this thesis, and in order to provide a solid base for the development and testing of some of the above-stated modules and behaviors, two additional tools have been developed.

A new 3D visualization for the Basestation was successfully developed and integrated in the Basestation application. Additionally, more information is displayed to help in the debugging process, namely the height maps, which would be very difficult to implement using basic OpenGL functions. Moreover, it is now possible to represent and visualize the ball in the 3D space, instead of just on the field plane.

Augmented Reality has been also introduced for the first time in the CAMBADA software. A new module and application was developed to explore this area and conduct the first experiments, meeting all the requirements set for both possible applications: the demonstration for audience in the robotics venues and also a new way of interacting with

the robots, by using a camera as a pointer to the virtual world.

In conclusion, all the developed work has been included in the master source-code branch and used in IranOpen14 and Robotica2014 competitions, which helped the team reach the finals and be awarded two honorable second places.

6.2 Future Work

Although positive results have been achieved, there is always margin for improvements and in this section, a few suggestions are left for possible future work.

In the sequence of the work done on the new Software Agent Architecture, it might be useful to separate the Invocation and Commitment conditions from the Behaviors. Behaviors should hold the algorithms to perform tasks and when assigning a `Behavior` to an `Arbitrator`, a Condition should also be linked to the Behavior. This way, the same Behavior could be easily used with different conditions on distinct `Roles`.

In free-play passes, the `Striker` currently relies on the good positioning of the receiver, performing a pass when they are almost facing each other. The receiver should also compute the same angular cost map calculated by the `Striker` in order to try to position himself in the direction where the `Striker` will be facing.

The work done on the `BKickToTheirGoal` to improve aiming with movement was based on experimental data, but the influence of the linear y-velocity of the robot should also be taken into account to calculate deviation angle from the robot's front direction. It would also be interesting to compare the results with a more analytical approach that computes and uses the instantaneous linear velocity of the ball, instead of odometry data from the robot.

A development tool for `HeightMaps` would also be useful. Currently, the agent code needs to be changed, compiled and run to be able to see the `HeightMap` in the Basestation. It could be interesting to have an application that could have a code editor and a visualization (or simulator) side by side to help in the development of new `HeightMaps` from scratch.

From the agent's perspective, the `HeightMap` class could also be used for the rest of the team positioning during gameplay, trying to block possible passes. Delaunay Triangulation provides an easy way to change formation but used in combination with height maps could lead to a very promising outcome.

Bibliography

[1] Hiroaki Kitano, Minoru Asada, Yasuo Kuniyoshi, Itsuki Noda, and Eiichi Osawa. Robocup: The robot world cup initiative. In *The First International Conference on Autonomous Agent (Agents-97)*, pages 340–347, 1997.

[2] RoboCup Logistics Official Website. http://www.robocup-logistics.org/, accessed 19-Jun-2014.

[3] RoboCup@Work Official Website. http://www.robocupatwork.org/, accessed 1-June-2014.

[4] H. Kitano, S. Tadokoro, I. Noda, H. Matsubara, T. Takahashi, A. Shinjou, and S. Shimada. RoboCup Rescue: search and rescue in large-scale disasters as a domain for autonomous agents research. In *Proc. of IEEE SMC '99 Conference on Systems, Man, and Cybernetics*, volume 6, pages 739–743, 1999.

[5] RoboCup Junior Official Website. http://www.robocup.org/robocup-junior/, accessed 30-May-2014.

[6] Thomas Wisspeintner, Luca Iocchi, Tijn Van Der Zant, and Stefan Schiffer. Scientific Competition and Benchmarking for Domestic Service Robots. *Interaction Studies*, 10(3):392–426, 2009.

[7] TechUnited AMIGO 2012 Team Description Paper. http://www.techunited.nl/media/files/definitieve_tdp.pdf, accessed in 21-Jun-2014.

[8] A. Neves, J. Azevedo, N. Lau B. Cunha, J. Silva, F. Santos, G. Corrente, D. A. Martins, N. Figueiredo, A. Pereira, L. Almeida, L. S. Lopes, and P. Pedreiras. *CAMBADA soccer team: from robot architecture to multiagent coordination*, chapter 2, pages 19–45. I-Tech Education and Publishing, Vienna, Austria, January 2010.

[9] L. Almeida, F. Santos, T. Facchinetti, P. Pedreiras, V. Silva, and L. S. Lopes. Coordinating distributed autonomous agents with a real-time database: The CAMBADA project. In *Proc. of the 19th International Symposium on Computer and Information Sciences, ISCIS 2004*, volume 3280 of *Lecture Notes in Computer Science*, pages 878–886. Springer, 2004.

[10] J. L. Azevedo, B. Cunha, and L. Almeida. Hierarchical distributed architectures for autonomous mobile robots: a case study. In *Proc. of the 12th IEEE Conference on Emerging Technologies and Factory Automation, ETFA2007*, pages 973–980, Greece, 2007.

[11] V. Silva, R. Marau, L. Almeida, J. Ferreira, M. Calha, P. Pedreiras, and J. Fonseca. Implementing a distributed sensing and actuation system: The CAMBADA robots case study. In *Proc. of the 10th IEEE Conference on Emerging Technologies and Factory Automation*, volume 2, pages 781–788, Catania, Italy, September 2005.

[12] F. Santos, L. Almeida, and L. S. Lopes. Self-configuration of an adaptive tdma wireless communication protocol for teams of mobile robots. In *Proc. of the 13th IEEE International Conference on Emerging Technologies and Factory Automation, ETFA 2008*, Hamburg, Germany, Sept. 2008.

[13] Frederico Santos, Luis Almeida, Luis Seabra Lopes, José Luís Azevedo, and Manuel Bernardo Cunha. Communicating among robots in the robocup middle-size league. In *RoboCup 2009: Robot Soccer World Cup XIII*, Lecture Notes in Artificial Intelligence. Springer, 2009.

[14] N. Figueiredo, A. J. R. Neves, N. Lau, A. Pereira, and G. Corrente. Control and monitoring of a robotic soccer team: The base station application. In *Proc. of the 4th International Workshop on Intelligent Robotics, IROBOT'09*, volume 5816 of *Lecture Notes in Computer Science*, pages 299–309, Aveiro, Portugal, 2009. Springer.

[15] Antonio J. R. Neves, Armando J. Pinho, Daniel A. Martins, and Bernardo Cunha. An efficient omnidirectional vision system for soccer robots: from calibration to object detection. *Mechatronics*, 21(2):399–410, March 2011.

[16] R. Dias, A. J. R. Neves, J. L. Azevedo, B. Cunha, J. Cunha, P. Dias, A. Domingos, L. Ferreira, P. Fonseca, N. Lau, E. Pedrosa, A. Pereira, R. Serra, J. Silva, P. Soares, and A. Trifan. CAMBADA 2013 Team Description Paper. `http://robotica.ua.pt/CAMBADA/docs/qualif2013/CAMBADA-tdp-2013.pdf`, accessed in 26-Jun-2014.

[17] R. Dias, F. Amaral, J. L. Azevedo, R. Castro, B. Cunha, J. Cunha, P. Dias, N. Lau, C. Magalhães, A. J. R. Neves, A. Nunes, E. Pedrosa, A. Pereira, J. Santos, J. Silva, and A. Trifan. CAMBADA 2014 Team Description Paper. `http://robotica.ua.pt/CAMBADA/docs/qualif2014/CAMBADA-tdp-2014.pdf`, accessed in 26-Jun-2014.

[18] R. Dias, F. Amaral, J. L. Azevedo, R. Castro, B. Cunha, J. Cunha, P. Dias, N. Lau, C. Magalhães, A. J. R. Neves, A. Nunes, E. Pedrosa, A. Pereira, J. Santos, J. Silva, and A. Trifan. CAMBADA 2014 Team Description Paper - IranOpen 2014.

[19] Michael Wooldridge and Paolo Ciancarini. Agent-oriented software engineering: The state of the art. In *Proc. of the First International Workshop on Agent-Oriented Software Engineering*, pages 1–28, Secaucus, NJ, USA, 2001. Springer.

[20] R.A. Brooks. A robust layered control system for a mobile robot. *Robotics and Automation, IEEE Journal of*, 2(1):14–23, March 1986.

[21] Georges Giralt, Raja Chatila, and Marc Vaisset. An integrated navigation and motion control system for autonomous multisensory mobile robots. In *Autonomous Robot Vehicles*, pages 420–443. 1990.

[22] R. Chatila and J. Laumond. Position referencing and consistent world modeling for mobile robots. In *Proc. of the IEEE International Conference on Robotics and Automation*, volume 2, pages 138–145, Mar 1985.

[23] Pattie Maes. The dynamics of action selection. In *Proc. of 11th International Joint Conference on Artificial Intelligence*, volume 2, pages 991–997, 1989.

[24] R. James Firby. An investigation into reactive planning in complex domains. In *Proc. of the Sixth National Conference on Artificial Intelligence*, volume 1 of *AAAI'87*, pages 202–206. AAAI Press, 1987.

[25] Michael P. Georgeff and Amy L. Lansky. Reactive reasoning and planning. In *Proc. of the Sixth National Conference on Artificial Intelligence*, volume 2 of *AAAI'87*, pages 677–682. AAAI Press, 1987.

[26] Peter Stone and Manuela Veloso. Multiagent systems: A survey from a machine learning perspective. *Autonomous Robots*, 8:345–383, 1997.

[27] Gerhard Weiss. *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*. The MIT Press, 2000.

[28] Adrian K. Agogino and Kagan Tumer. A multiagent approach to managing air traffic flow. *Journal of Autonomous Agents and Multi-Agent Systems*, 23(1):1–25, 2010.

[29] TechUnited 2014 Team Description Paper. `http://www.techunited.nl/media/files/TDP2014.pdf`, accessed in 21-Jun-2014.

[30] Stefan Triller. A cooperative behaviour model for autonomous robots in dynamic domains. Master's thesis, University of Kassel, February 2009.

[31] Sascha Lange, Christian Muller, and Stefan Welker. Behavior-based approach, November 2008.

[32] Martin Lauer, Roland Hafner, Sascha Lange, and Martin Riedmiller. Cognitive concepts in autonomous soccer playing robots. *Cognitive Systems Research*, 11(3):287–309, 2010.

[33] Anand S. Rao and Michael P. Georgeff. Modeling rational agents within a BDI-architecture. In *Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning*, pages 473–484. Morgan Kaufmann publishers Inc.: San Mateo, CA, USA, 1991.

[34] Max Risler. *Behavior Control for Single and Multiple Autonomous Agents Based on Hierarchical Finite State Machines.* Fortschritt-berichte vdi, Technische Universitat Darmstadt, May 15 2009.

[35] Hidehisa Akiyama. HELIOS2007 Team Description Paper, 2007.

[36] Sérgio Martins. Treinador automático para a equipa de Futebol Robótico CAMBADA. Master's thesis, University of Aveiro, 2013.

[37] João Cunha, Rui Serra, Nuno Lau, Luís Seabra Lopes, and António J. R. Neves. Learning robotic soccer controllers with the Q-Batch update-rule. In *IEEE International Conference on Autonomous Robot Systems and Competitions (ICARSC)*, pages 134–139, May 2014.

[38] João Silva. Sensor fusion and behaviours for the CAMBADA Robotic Soccer Team. Master's thesis, University of Aveiro, 2008.

[39] Ricardo Sequeira. Strategic coordination of cambada robotic soccer team. Master's thesis, University of Aveiro, 2009.