Technical University of Crete, Greece
School of Electrical and Computer Engineering

Soccer Player Behavior Development for the RoboCup Standard Platform League

Helen Tsagkarogianni
etsagarogianni at isc tuc gr

Thesis Committee
Associate Professor Michail G. Lagoudakis (ECE)
Associate Professor Georgios Chalkiadakis (ECE)
Dr. Nikolaos I. Spanoudakis (PEM)
Chania, June 2019

Πολυτεχνειο Κρητης

Σχολη Ηλεκτρολογων Μηχανικων Και Μηχανικων Υπολογιστων

Ανάπτυξη Συμπεριφοράς Ποδοσφαιριστή για το Πρωτάθλημα Standard Platform του RoboCup

Ελένη Τσαγκαρογιάννη

etsagarogianni at isc tuc gr

Εξεταστική Επιτροπή

Αναπληρωτής Καθηγητής Μιχαήλ Γ. Λαγουδάκης (ΗΜΜΥ)

Αναπληρωτής Καθηγητής Γεώργιος Χαλκιαδάκης (ΗΜΜΥ)

Δρ. Νικόλαος Ι. Σπανουδάκης (ΜΠΔ)

Χανιά, Ιούνιος 2019

Abstract

Humans are used to operate in dynamic environments, which nevertheless can be quite unpredictable and complicated for a robot. Therefore, for an autonomous robot to be able to adapt and act in such an environment, it must tackle a large variety of the problems, using a top-down approach that breaks them down into simpler ones. The annual robot soccer (RoboCup) competition offers such a dynamic environment and one of the main components of any RoboCup player is the algorithm responsible for decision making, known as robot behavior control. In this thesis, we aim to develop from scratch soccer player behaviors for the Standard Platform League (SPL), where all teams share the same robot platform, but develop different software, in an attempt to better understand the process of behavior development. Behavior development can be very tricky, since it combines information from different modules (vision, motion, communication, estimation, etc.) and decides the actions that need to be taken for the agent to achieve its goal. We worked on a recent code base framework developed by SPL Team B-Human that uses the C-based Agent Behavior Specification Language (CABSL) for the behavior development, which is freely available, however with all behavior code turned to void on purpose. We implemented three soccer behaviors for three basic soccer roles: a striker, a defender and a goalkeeper. Furthermore, we implemented an additional behavior, called ball control, in order to assess the utilization of the framework beyond typical robotic soccer behaviors. We aim for this work to serve as the initial steps for new code base of our SPL team Kouretes and as a guide for assisting the understanding and the use of the chosen framework and its capabilities.

Περίληψη

Οι άνθρωποι είναι συνηθισμένοι να λειτουργούν σε δυναμικά περιβάλλοντα, τα οποία ωστόσο μπορούν να είναι αρκετά απρόβλεπτα και περίπλοκα για ένα ρομπότ. Επομένως, για να μπορεί ένα αυτόνομο ρομπότ να προσαρμόζεται και να δρα σε ένα τέτοιο περιβάλλον, πρέπει να αντιμετωπίσει μια μεγάλη ποικιλία των προβλημάτων, χρησιμοποιώντας μια προσέγγιση top-down , η οποία τα διαιρεί σε απλούστερα. Ο ετήσιος διαγωνισμός ρομποτικού ποδοσφαίρου (RoboCup) προσφέρει ένα τέτοιο δυναμικό περιβάλλον και ένα από τα κύρια συστατικά του κάθε παίκτη είναι ο αλγόριθμος που φροντίζει για τη λήψη αποφάσεων, γνωστός ως έλεγχος ρομποτικής συμπεριφοράς. Στην παρούσα διπλωματική εργασία, στοχεύουμε να αναπτύξουμε από το μηδέν τις συμπεριφορές ενός ποδοσφαιριστή για το πρωτάθλημα Standard Platform League (SPL), όπου όλες οι ομάδες μοιράζονται την ίδια ρομποτική πλατφόρμα, αλλά αναπτύσσουν διαφορετικό λογισμικό, σε μια προσπάθεια να κατανοήσουμε καλύτερα τη διαδικασία ανάπτυξης συμπεριφοράς. Η ανάπτυξη συμπεριφοράς μπορεί να αποβεί δύσκολη, καθώς συνδυάζει πληροφορίες από διαφορετικές ενότητες (όραση, κίνηση, επικοινωνία, εκτίμηση, κλπ.) και αποφασίζει τις ενέργειες που πρέπει να εκτελεσθούν για την επίτευξη του στόχου του πράκτορα. Δουλέψαμε σε ένα πρόσφατο πλαίσιο λογισμικού που αναπτύχθηκε από την ομάδα SPL Team B-Human που χρησιμοποιεί τη γλώσσα C-based Agent Behavior Specification Language (CABSL) για την ανάπτυξη συμπεριφοράς, το οποίο είναι ελεύθερα διαθέσιμο, ωστόσο με όλο τον κώδικα συμπεριφοράς διαγραμμένο. Αναπτύξαμε τρεις συμπεριφορές παικτών για τρεις βασικούς ρόλους ποδοσφαίρου: έναν επιθετικό, έναν αμυντικό και ένα τερματοφύλακα. Επιπλέον, αναπτύξαμε μια πρόσθετη συμπεριφορά, που ονομάζεται έλεγχος μπάλας, προκειμένου να εκτιμήσουμε την αξιοποίηση του πλαισίου λογισμικού, πέρα από τις τυπικές ρομποτικές συμπεριφορές ποδοσφαίρου. Στοχεύουμε το έργο αυτό να αποτελέσει τα αρχικά βήματα για τη δημιουργία της νέας βάσης λογισμικού της ομάδας μας SPL Κουρήτες και οδηγό για την καλύτερη κατανόηση και χρήση του επιλεγμένου πλαισίου και των δυνατοτήτων του.

Acknowledgements

I would like to thank my advisor Michail G. Lagoudakis for his guidance and support throughout the implementation of this thesis.

I would also like to thank my team Kouretes and especially Dimitris Chatziparaschis, George Apostolakis, and Nektarios Sfyris for their support.

Last but not least, I would like to thank my family for their love, patience and support.

Contents

List of Figures

Listings

Chapter 1

Introduction

Autonomous agents can be of a great assistance in numerous fields of our everyday lives; from providing aid in a household to heavy duty automation required in factories or high precision space excursions. A robotic autonomous agent combines the majority of the topics studied by Artificial Intelligence. Machine vision, networking, navigation provide the agent with the information required to make a decision, but doesn't lead to a decision by themselves. The algorithm that combines the above and leads the agent to its goal, in our case is called behavior.

The RoboCup competition aims to solve various research problems regarding the perception, localization, movement of the agents and coordination while aiming for robotic autonomy. The competition assists in creating objectives for the participating teams; it sets specific tasks that are accompanied by a variety of challenges and gradually get more complicated. In addition, since some of the RoboCup leagues revolve around soccer, a couple of things are intuitive such as the rules or tactics.

It is fascinating working with a robot, a system that combines hardware and software, that is capable of being autonomous, exists in the real world and can in a way alter it with its own actions. By working on a project like this, our goal is to gain experience and try to implement behaviors using a new tool in order to put into action what we have learned so far during our studies, but also better understand such a complex system.

Having limited resources and no prior experience in the field of programming autonomous agents in combination with trying out a new framework made the project's structure not intuitive. On top of that, the task of developing a fully autonomous soccer behavior made is quite complicated, making this implementation a challenge.

1.1 Thesis Contribution

We started our implementation with executing basic functions, such as scanning and locating the ball, aiming to better understand the software architecture. Then, we started adding more modules into our behaviors, controlling both the head and the body of the robot and also making our robot act based on its position in the field. After some experimentation with different operating systems and code releases, we achieved a good understanding of the project and the use of the modules given and we were able to utilize them to create basic soccer and soccer-like behaviors.

We managed to achieve a solid basic understanding of the B-Human toolkit and implemented four different behaviors using the chosen framework that were both functional in simulation and on a real robot on the field located at the Kouretes Lab. Also we believe that we can provide a basic guide for all the steps required into assisting any implementation of more sophisticated behaviors.

The work done in this thesis is aimed to be used as a starting step for our robotic team Kouretes. For that matter there is also a thorough guide in the appendix providing information on how to execute the implemented behaviors, as well as implementation tips.

1.2 Thesis Overview

In Chapter 2 we go over the background information required for this thesis. To be more precise, we go through an introduction to RoboCup, a brief overview of the Standard Platform League, a presentation of the humanoid robot Nao that was used as our agent and a description of the modules required and provided by the chosen software framework. In Chapter 3 we define the goals of this thesis and we review the different frameworks used by other participating teams over the years as well as the one we concluded on using and what led us to that choice. In Chapter 4 we describe the procedure we followed in order to understand the framework and thoroughly explain the implemented behaviors. In Chapter 5 we present the results of the implemented behaviors in both simulation and physical world. In Chapter 6 we conclude the thesis by discussing the proposed procedure and the implemented behaviors in addition to ideas for future work. In Appendix A we

have added a user guide that assists with the execution of the code and also provides all the information required for the reader to create new behaviors.

Chapter 2

Background

2.1 Multi-Agent Systems

An autonomous agent is an intelligent entity, of robotic or software nature able to adapt and learn based on sensory data in order to achieve a goal given by its creator. It has been given all the necessary information to carry out the given task without any further interference from humans. At the beginning of the learning process it might require some assistance, but past that point it should be able to learn and adapt in order to fulfill the given task. Its nature and complexity may vary from a robotic vacuum to self-landing space rockets. In this thesis, the goal of our agent is to play soccer meaning complying with the game rules and scoring a goal, in case of a striker role, or preventing a goal, in case of a defender or a goal keeper, in order to win the match.

Since the game of soccer is played between two teams, our autonomous agents are a part of a multi-agent system, where the robots communicate with each other and work together towards a common goal. The environment is considered competitive, since in order for one team to win, the other must lose. Also, the agents of the same team need to cooperate towards the common goals.

2.2 RoboCup

RoboCup [1] is an international robotics competition founded in 1996 by a scientific group of university professors. It occurs annually and it aims to promote the research of robotics

and Artificial Intelligence by making easily accessible to the public but in the same time progressively harder each year for the participating teams. The reason it revolves around soccer is that a lot of issues need to be resolved in order to produce a successful team. To be more precise, a robotic team needs to be able to tackle the difficulties that a real life environment with dynamic lightning, uneven terrain, such as artificial grass, can create as well as opponent teams with different strategies. Not to mention its original mission was to develop a team of robots that would be capable of competing and winning the World Cup human champion team by 2050. Today, RoboCup hosts a great variety of leagues each one with its own subcategories.

- RoboCupSoccer
- RoboCupRescue
- RoboCup@Home
- RoboCupIndustrial
- RoboCupJunior

2.2.1 RoboCupSoccer

RoboCupSoccer is divided into five different leagues:

Humanoid League: The participating teams are composed of humanoid robots with human like sensors. The teams aren't restricted in one particular platform but the sensors used are strictly human like. A few of the robots participating can be seen in Figure 2.1.

Standard Platform League: The teams competing in this league use fully autonomous humanoid robots but all make use of the same platform, the humanoid robot Nao of the Softbank Robotics. The field set up complicates the challenge even further and provides room for new challenges each year. In Figure 2.8 we can see a snapshot of a SPL match.

Middle Size League: Teams competing into the middle size league are free to design their own hardware but there are maximum size and weight limitations of their autonomous robots as seen in Figure 2.2.

Figure 2.1: RoboCup Humanoid League

https://www.robocup.org

Figure 2.2: Middle Size League

https://www.robocup.org

Small Size League: One of the oldest RoboCup leagues. The teams are restricted to specific dimensions but the robots are not autonomous since the processing required for coordination as well as the control of the robots is executed on a computer outside the field. A kick off can been seen in Figure 2.3.

Figure 2.3: Small Size League

https://www.robocup.org

Figure 2.4: Simulation League

https://www.robocup.org

SimulationLeague: Another one of the oldest leagues focusing for on the artificial intelligence and the implementation of a team strategy. A snapshot of a simulation match can be seen in Figure 2.4

Figure 2.5: RoboCup Rescue League

https://www.robocup.org

2.2.2 RoboCupRescue

RoboCupRescue focuses both in real life robots (Figure 2.5) and simulation (a gazebo snapshot in Figure 2.6) in order to implement advanced robotic solutions to tackle emergency scenarios.

Robot: The teams competing in this league are faced with a lot of challenges that a unknown environment can create in order to search and rescue individuals. Each solution is objectively evaluated taking into consideration the scenario in which the robot is placed but also promotes the collaboration between the teams.

Simulation: This league instead of only the focus of intelligent agents that are able to respond to a disaster scenario, it also aims on the development of simulators that are capable to accurately simulate such scenarios as well as emulate realistic disaster phenomena.

Figure 2.6: RoboCup Rescue Simulation League

https://www.robocup.org

2.2.3 RoboCup@Home

RoboCup@Home focuses on the development of robots that aim to provide services and assistance to a household.

Open Plataform: In this league every custom platform is allowed.

Domestic Standard Platform: Teams competing are restricted by the given platform.

Social Standard Platform: Same as Standard Platform League, teams competing are restricted in the use of the same robot this time Pepper of Softbank Robotics.

2.2.4 RoboCupIndustrial

RoboCupIndustrial is one of the newest competitions where the robots developed are intended for work and industrial usage.

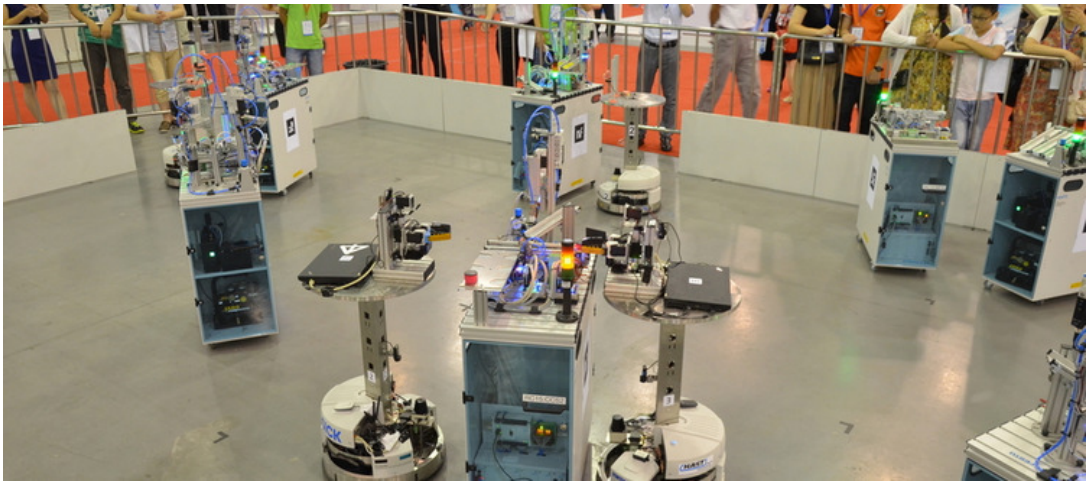


Figure 2.7: RoboCup Logistics League

https://www.robocup.org

RoboCup@Work: The newest league in RoboCup that utilizes concepts from the rest of the RoboCup competitions aiming to tackle challenges in industrial and service robotics in order to enable their use in work-related scenarios.

Logistics: An application driven league inspired by the industrial scenario of a smart factory, where a number of machines are responsible for all the stages of manufacturing a final product. From refinery to assembly and even product modifications. A snapshot can be seen in Figure 2.7

2.2.5 RoboCupJunior

RoboCupJunior offers several challenges, each emphasizing both cooperative and competitive aspects in order to provide an exciting introduction to the field of robotics through hands-on experience. Focusing on educating its competitors it also sponsors local, regional and international robotic events for young students. It 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.

Soccer: 2-on-2 teams of autonomous mobile robots play in a highly dynamic environment, tracking a special light-emitting ball in an enclosed, land marked field.

OnStage: One or more robots come together with humans, dressed in costume and moving in creative, interactive and collaborative ways.

Rescue: Robots identify victims within re-created disaster scenarios, varying in complexity from line-following on a flat surface to negotiating paths through obstacles on uneven terrain.

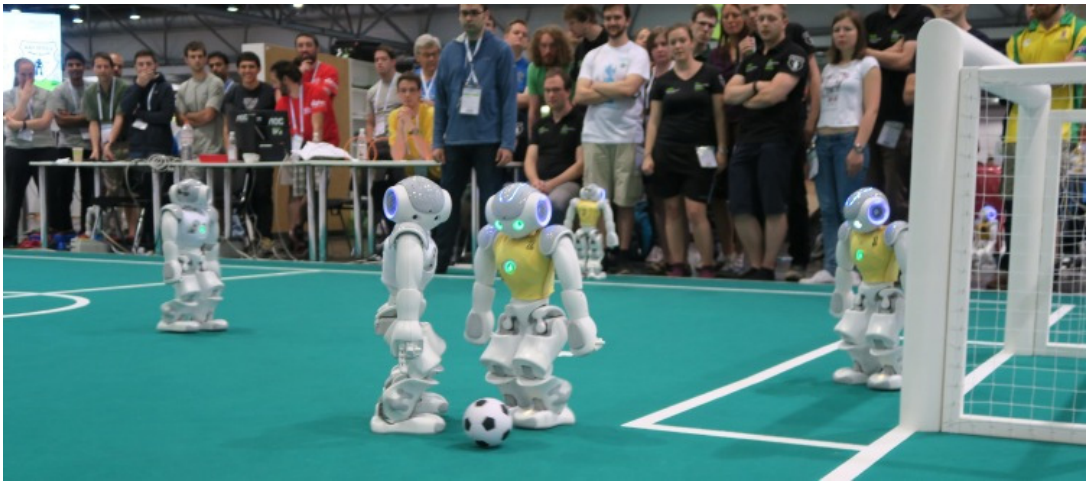


Figure 2.8: Standard Platform League

https://www.robocup.org

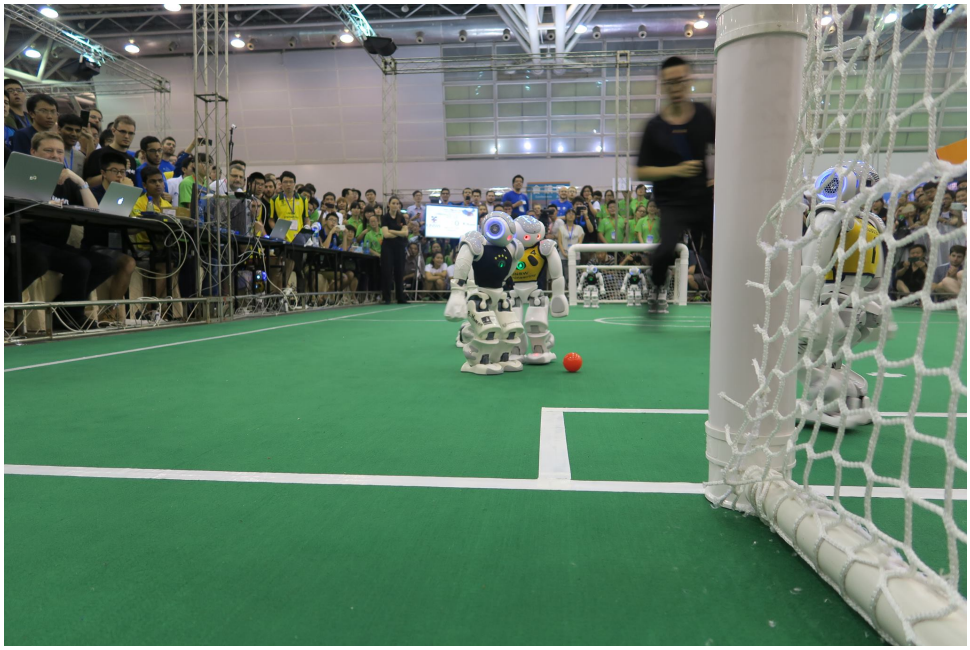
2.3 RoboCup Standard Platform League

Standard Platform means that the for the implementation platform the software, mostly operating system, and the hardware are defined and not created by the teams themselves. In a way guarantees equal footing and concentrates the efforts on the other parts of the implementation and not in the creation of the robotic system. Each year, the committee updates the rules creating new challenges for the teams to overcome. Additionally, allowing the cooperation between teams in terms of being able to use the code created by other teams as long as you make changes and improve it, is aiming in the advancement of all the community. Such changes can be from the color of the ball to the location and the characteristics of the field. We are able to see a few of the visual changes over the years though the Figures 2.9, 2.10, 2.11, 2.12 and 2.13 but major changes occur in other aspect of the competition as well, such as the rules.

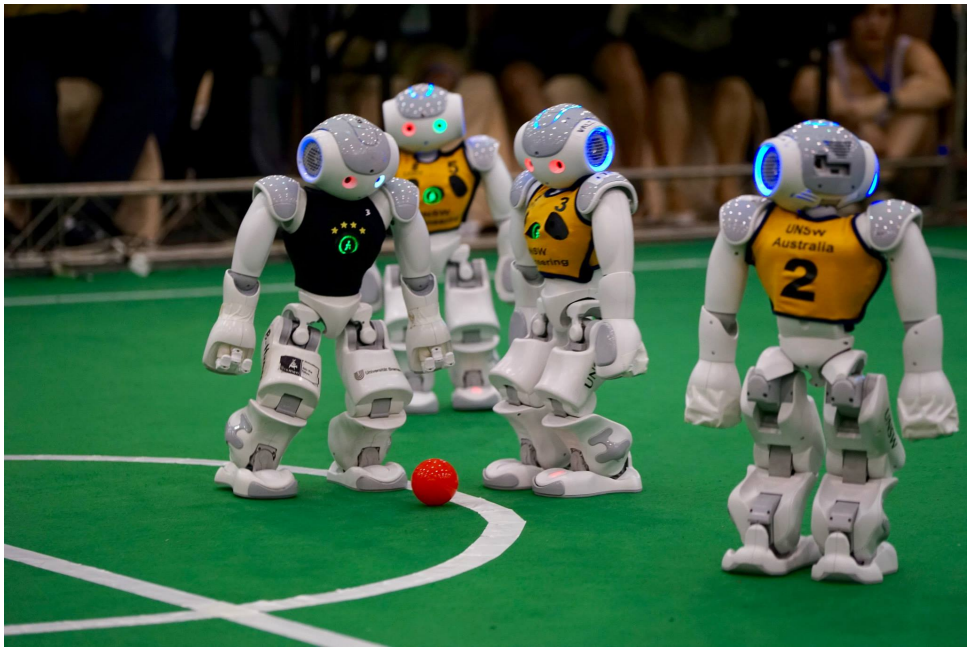
(a) (b)

(c)

Figure 2.9: Standard Platform League 2008 was the first year that Nao robots were used. We see teams composed by two robots, an orange ball, different color goalposts, artificial lighting and no other fields close. *Source:* http://robots.newcastle.edu.au/robogallery.html



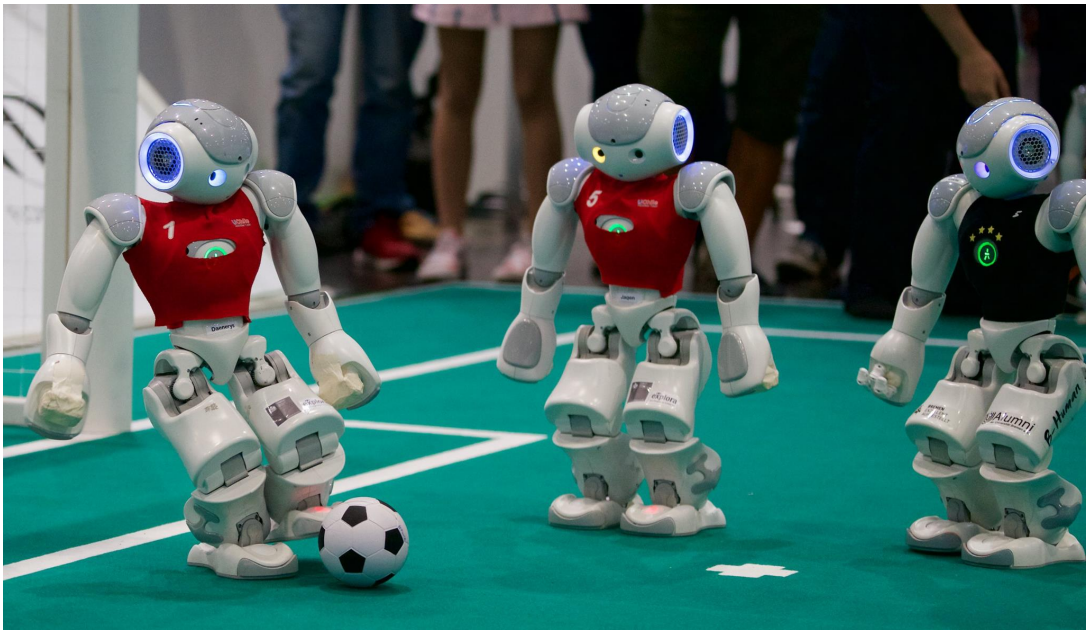
(a)



(b)

Figure 2.10: Standard Platform League 2015 B-Human vs rUNSWift. *Source:* https://www.facebook.com/RoboCupSPL/

(a)



(b)

Figure 2.11: Standard Platform League 2016 Indoor field and quarter final between B-Human and Chile. *Source:* https://www.facebook.com/RoboCupSPL/

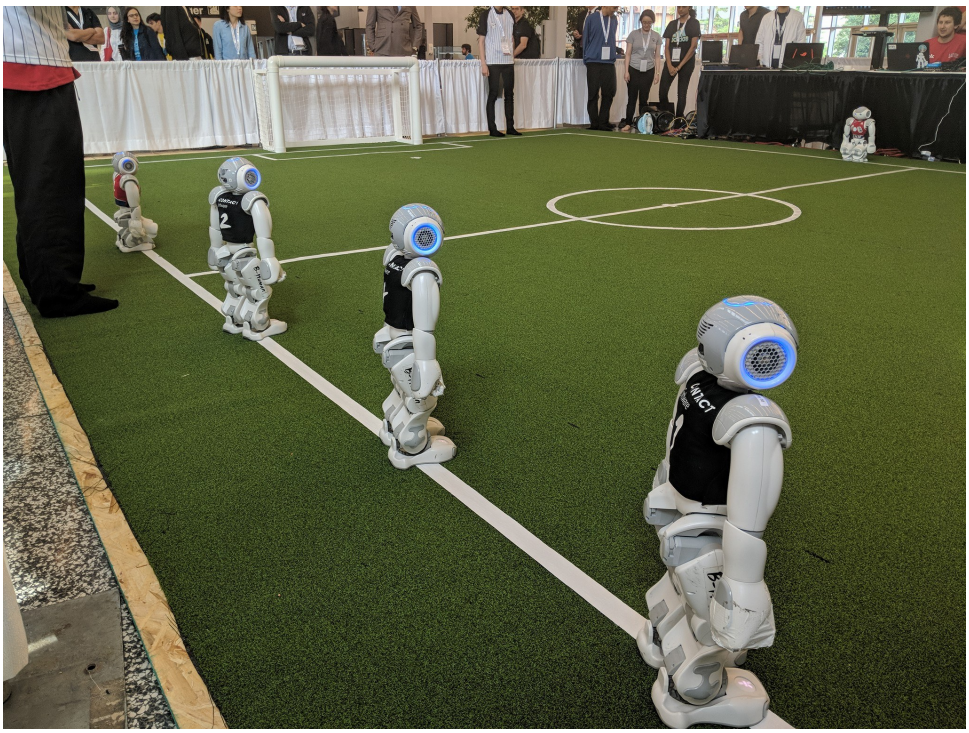
(a)

(b)

Figure 2.12: Standard Platform League 2016 Outdoor field outdoor and Finals between B-Human and NaoDevils.

Source: https://www.facebook.com/RoboCupSPL/

(a)



(b)

Figure 2.13: Standard Platform League 2018 outdoor field.

Source: https://www.facebook.com/RoboCupSPL/

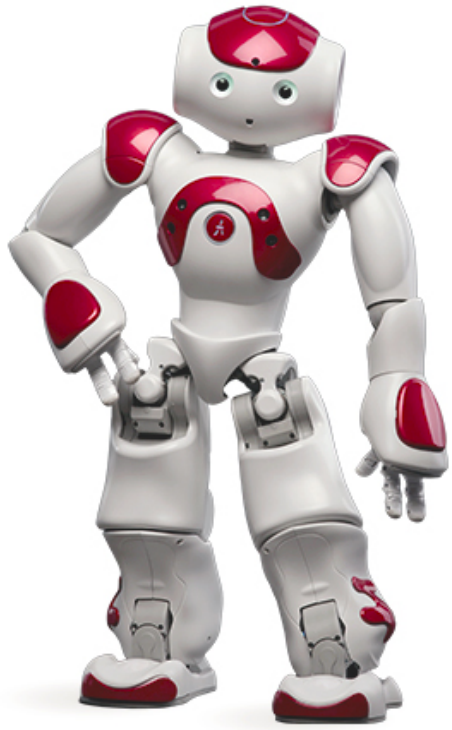


Figure 2.14: Model of Nao V5

http://doc.aldebaran.com

2.4 Nao

The platform for this competition is the humanoid robot Nao developed by SoftBank Robotics previously known as Aldebaran Robotics. A robot that can act as an autonomous agent. In this thesis we are using a Nao version 5 with Naoqi 2.1.0.19 seen in Figure 2.14. Nao V5 has 25 Degrees Of Freedom (DOF). An overview of its characteristics can be seen in Figure 2.15. With the installation of the B-Human libraries and modules a few procedures of Naoqi software are being deactivated.

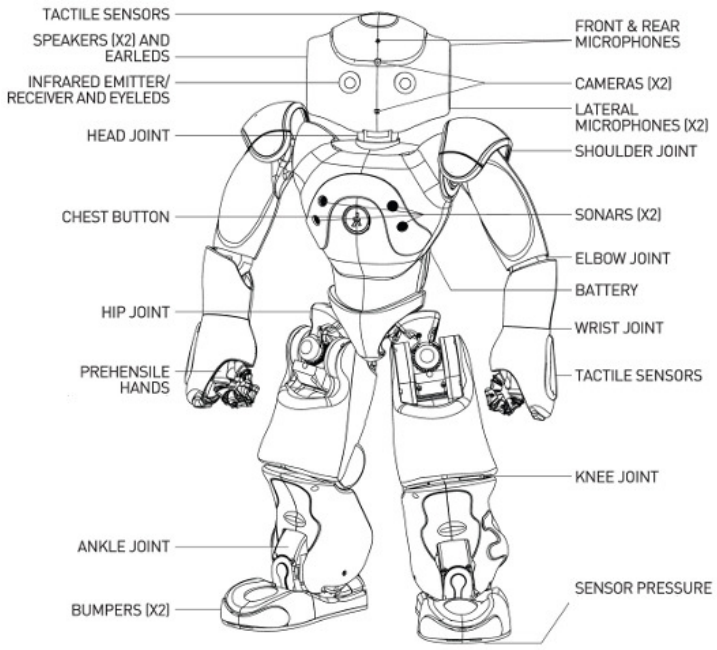


Figure 2.15: Nao V5 characteristics

http://doc.aldebaran.com

2.5 B-Human Project

B-Human is a RoboCup team of the University of Bremen and the German Research Center for Artificial Intelligence (DFKI). The team was founded in 2006 as a team in the Humanoid League, but switched to participating in the Standard Platform League in 2009.

In this thesis, we make use of the system designed by B-Human that allows us to focus on the implementation of the behaviors since the rest of the infrastructure is already implemented. To be more precise, this system provides us with all the required modules that control the cognition and motion but is stripped of the Behavior control. We will briefly go through what has been provided by the their code release of 2017 [2] and 2018 [3].

2.6 Robotic Cognition

We refer to cognition when our robot is able to sense and understand its surroundings via sensors, cameras, contact and touch sensors. In addition, it should be able to create beliefs regards the environment, generate a map of its surroundings, locate its position inside the created map as well as remember previous routes and locations. Based on that information it decides to act in order to achieve a specific goal.

2.6.1 Machine Vision

The term Machine Vision encapsulates all the technologies and methods used in order to extract useful information from an image. The extracted information can be in various forms depending on the application. In our case, the data are extracted from the two cameras of Nao and have a YUV422 format that is then converted to ECImage. The cameras are positioned as shown in Figures 2.16 and 2.17.

Since there is not enough overlap for stereo video image and the model of the Nao used does not support simultaneous image record nor simultaneous maximum resolution, a few compromises are being made stated in the B-Human code release of 2017 [2].

2.6.2 Networking and Communication

Since soccer is a team based game, communication between the robots of the same team is vital in order to transmit messages about their perception, localization beliefs as well as team tactics. For that to be achieved, a network must be established where all the robots, from different teams, are able to communicate without their messages being intercepted or mixed. This is achieved with the use of a magic number. Each team has its own magic number and robots with different magic numbers will ignore each other. In the behavior implementation we make use of variables that are populated with the use of all the beliefs of the different robots such as `teamBallModel`, an object that we use in our behaviors which stores the coordinates of the ball relative to the field.

2.6.3 Navigation

This project provides us with modules that handle both universal navigation based on the map and the landmarks of the soccer field and reflex navigation in order to avoid collisions.

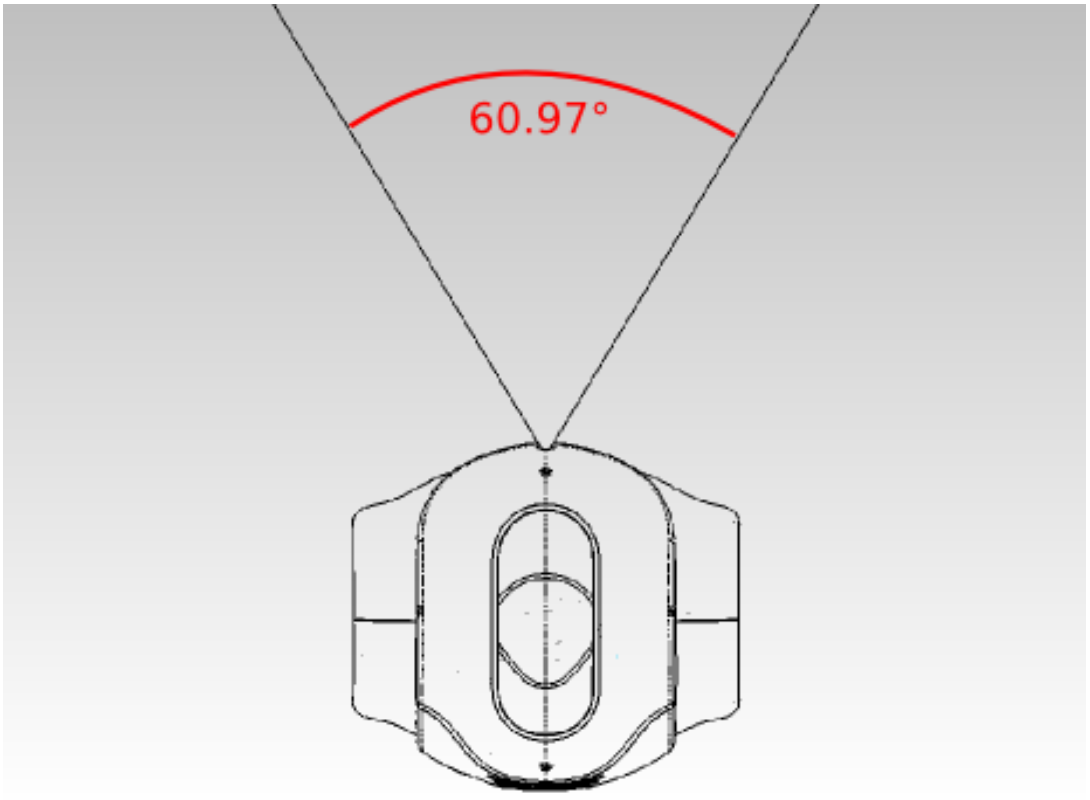


Figure 2.16: Nao camera top view

http://doc.aldebaran.com

In order to focus mostly on the behavior implementation we make use only of the universal navigation based on the mapping with the field characteristics and dimensions.

Mapping

Since the dimensions of the field are static and are given via a configuration file there is no need for active mapping. The map is in a way predefined.

Localization

Localization is the procedure where the agent resolves questions regarding its location. Since the agent is aware about the dimensions of the field as well as the position of the landmarks, the localization occurs based on these features. Based on that, checking and

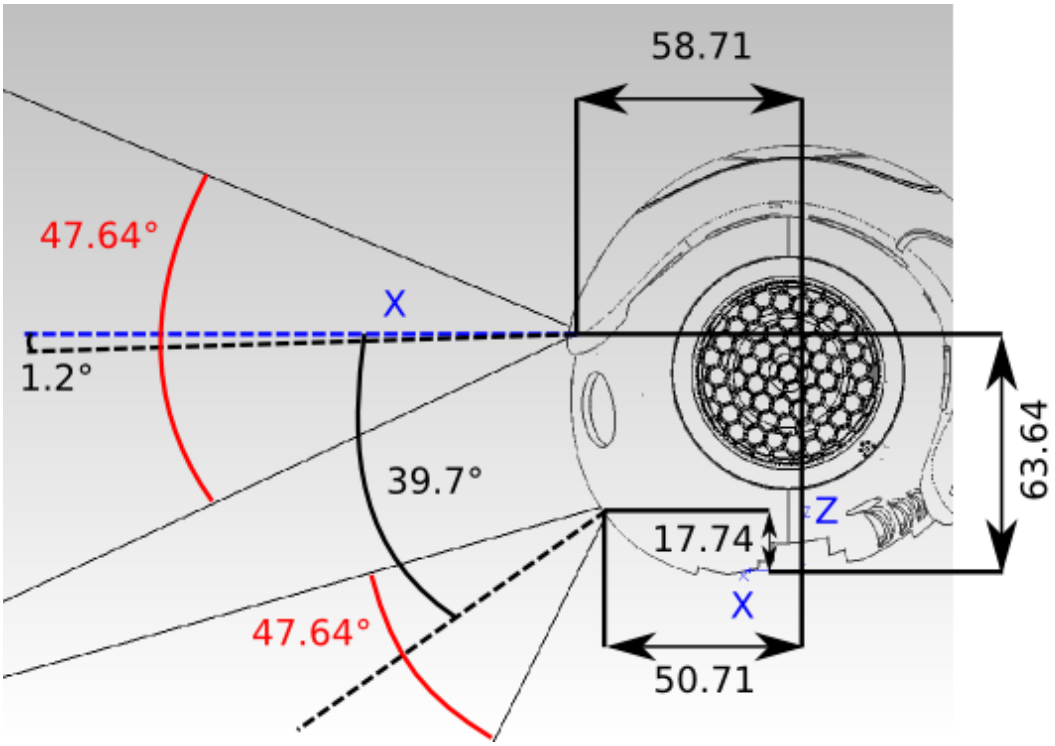


Figure 2.17: Nao camera side view

http://doc.aldebaran.com

validating their position greatly reduces the localization error.

Path Planning

The path planner module is responsible for handling long walking distances to a target that would probably end into a deadlock if the robot was just using reactive control. In this project, B-Humans use a visibility-graph-based 2-D A* planner but it is not included into our implemented behaviors.

Motion Control

The motion control module is responsible for the movement of the agent. It supplies ROS[1] with the required angles and speed in order to execute any motion from a kick to a walk

[1]Robot Operating System

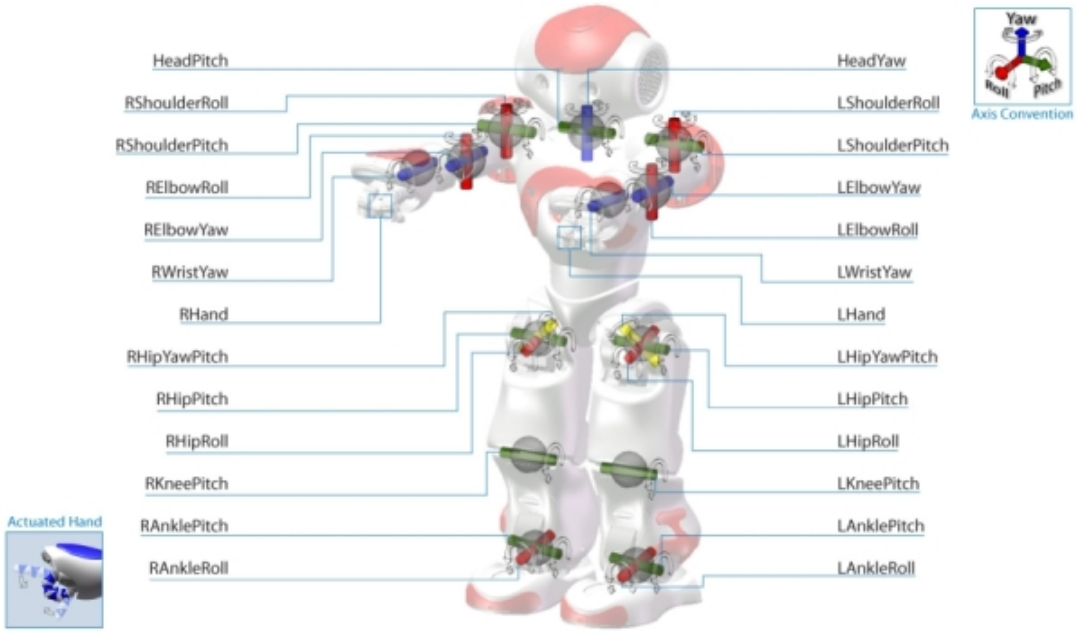


Figure 2.18: Nao's joint view

http://doc.aldebaran.com

motion as well as head motions and any other blocking motion. There are several engine responsible for the different motions in this approach. The `WalkingEngine` is responsible for the walk and stand motions as well as some special InWalk kicks. The KickEngine is responsible for the generation of any other kick. The GetUpEngine is responsible for figuring out the best way for the robot to get up in case of a fall and also to make sure that during a fall the damage inflicted on the robot is be limited. In Figure 2.18 we can see all the different joints of Nao with their movement axes that are controller by the different engines.

2.6.4 Behavior Control

The behavior control is responsible for the decisions our agent makes in order to achieve its task. Given the information of all the modules mentioned above, behavior control combines them and concludes to the action required while sending the appropriate requests to motion control and all the interactive modules.

Chapter 3

Problem Statement

The algorithm that is responsible for the decisions that our agent makes, its behavior, in order to achieve its goal can be very complicated. For implementing such complicated behaviors there must be a solid project architecture with a reliable framework. In this last section of this chapter we are going to examine briefly what frameworks have been used in the past and present for soccer behavior development. Since the goal of this thesis is behavior implementation, we did not focus on the development of a new framework.

3.1 Thesis Objectives

The objective of this thesis is the implementation of three soccer behaviors and one soccer-like behavior aiming to better understand behavior development, but also utilize the chosen framework beyond the soccer domain.

The three soccer behaviors to be implemented are: (a) a striker, (b) a defender and (c) a goalkeeper. The role of the striker aims to dribble the ball and ultimately score a goal against the opponent team, while avoiding the opponent defenders. The role of the defender aims to defend its own side of the field from a potential attack of the opponent strikers and kick the ball away towards towards the opponent half of the field. The role of goalkeeper is to guard its goal from any threatening kick and prevent the opponent from scoring a goal, while kicking the ball away from its own goal area.

The last behavior we aim to implement is an abstract behavior aiming to control the ball in order to keep it inside a designated target area, which in our case is chosen to be the center circle of the field. While this behavior might not be useful for robot soccer

playing, it combines different walking approaches to the ball, depending on the current robot orientation with respect to the ball and the center of the field.

3.2 Framework Selection

The code of Kouretes along with all the behaviors and most of the frameworks where implemented to work with Nao v3.3. Experimenting with different behaviors, algorithms for the university courses, playing soccer in addition to the passage of time had great impact on the team's robots resulting into the slow breakdown of the older Nao that needed to be replaced with new ones. The development of the robot Nao also advanced as well resulting in newer Nao versions with a lot of differences. The Nao v4 and v5, while similar in appearance are quite different in both hardware and software, NAOqi 1.14 for Nao V4 and V3 while Nao V5 has the 2.1 version of NAOqi, the already implemented code must be reworked and converted to the newest version in order to work. A few efforts were made in the past but as a team we concluded that it might be better to start all over again with the code base of B-Human. Hence we started experimenting with CodeRelease 2017 creating a few behaviors and trying to understand the code. Then, as we will see in the next chapters we transitioned what was already implemented into the CodeRelease 2018 when it was released since we concluded that it has useful additions and we had not yet a great load of code to transition to the different code base.

3.3 Related Work

The SPL teams[1] use a variety of tools in order to implement their soccer behaviors. Lets briefly go over a few of them:

ASU RoboSoc Devils [4] have used Model Based Designed and were able to create simple behaviors for Nao with the combination of Simulink library for simulation and code generation, Statefrom interface API and C/C++ interface API.

[1] https://spl.robocup.org/teams/

Bembelbots Frankfurt [5] started with the use of a simple state machine approach and then transition to XABSL[1] in order to implement their behaviors. Since there were a few issues and the transition to CABSL[2] was almost seamless and easily integrated they decided to switch to CABSL and also started working on a stand alone behavior simulator. They aim to create a simulator that will assist with the implementation of new behaviors but also help the newest members of their team to quickly adapt.

Berlin United - Nao Team Humboldt [6] uses XABSL in order to model the behavior of individual robots as well as the whole team. They have also implemented a XabslEditor with the use of Java in order to assist with the development of such behaviors. This tool consists of a full featured editor with syntax highlighting, a graph viewer for visualization of behavior state machines and an integrated compiler. In the same time they work on the implementation of another tool, a visualizer for the XABSL execution tree in order to assist in the monitor the decisions process of the robot at the run time.

B-Human [2, 3] team used in the past XABSL in order to implement their behaviors but due to its programming restrictions they decided to implement our own behavior engine called SMBE[3] that was based on XABSL but resolved its issues. Through that effort they designed CABSL, a language that is designed to describe an agent's behavior as a hierarchy of state machines and they are using it as behavior framework ever since.

Camellia Dragons [7] team is making use of the B-Human code release of 2016 with extra modifications. They added five major functions, a motion module responsible for collective play, a realistic ball perception method, different team roles, a computational cost reduction method of self-localization system and a penalty kick behavior.

Cerberus [8] is a using a module Planner that is responsible of tracking the changes in the robot's environment and then decide on the next action. In its early stages it was a market based planner and a Dec-POMDP[4] based planner with the latest version being

[1]Extensive Agent Behavior Specification Language
[2]C-based Agent Behavior Specification Language.
[3]State Machine Behavior Engine
[4]Partially Observable Markov Decision Process.

based on a finite state machine.

Cuauhpipiltin - formerly Eagle Knights [9] is aiming to use a local motion planner the ND[1] in order to navigate towards the ball or the goal while avoiding collisions with other robots with combination of a multi-layered motion planning and control.

DAInamite [10] team implements behaviors with the use of hierarchical state machines written in Python and also modified the SimSpark3D simulator from Nao-Team Humboldt in order to test behaviors.

Dutch Nao Team [11] team while it started with their own Python framework, transitioned to Berlin United's code and then B-Human code base. Currently ended up implementing and using their own their framework since they wanted to be able to provide their team with a fully understandable codebase as well as extend documentation assisting both old and new members.

HULKs [12] team is developing their behaviors with the use of a C++ based MSM[2], a state machine framework from Boost libraries. They also divide their implementation into multiple state machines and switch between them with the use of a SMSwitcher.

Kouretes [13] team has used in the past the Agent Systems Engineering Methodology (ASEME) [14, 15] in order to compose the behavior of the robotic team. ASEME breaks down the behavior into smaller components called activities until a level of provided base activities is met. Members of our team also implemented their own CASE[3] tool called Kouretes Statechart Editor [KSE] [16] in order to assist the developer with the design of the state chart model either graphically from scratch or by first writing liveness formulas[4] [17] and transforming them automatically to an abstract state chart.

[1] Nearness Diagram
[2] Meta State Machine.
[3] Computer-Aided System Engineering
[4] The expressions that specify liveness properties, one of the two types of responsibilities of an agent.

MI-PAL [18] team uses a Base-Control architecture in order to build their behaviors which are encoded as LLFSMs[1] and can be considered as Augmented Timed Finite-State Machines similar to the original LISP-based behaviors of the subsumption architecture and the Toto robot. They also developed MiEditLLFSM in Java, an editor for the creation of their behaviors.

Nao Devils [19] are using the B-Human code base of 2015 while changing and adapting the code based on their needs. Their behaviors were implemented with the use of XABSL and later transitioned also to CABSL.

NTU RoboPAL [20] team uses FSMs to model their behaviors with multiple levels and different pools that provide the required roles, way points and skills.

SPQR [21] initially was using their own framework called OpenRDK but then transitioned to the framework developed by B-Humans and used it as a base in order to develop their code adding their own modules about perception, coordination, decision making and in general all the behavior. Currently they are making use of the 2017 framework.

Team-NUST [22] has implemented their own framework working with Naoqi and creating the required modules in order to implement their teams code. The behavior manager module is the one responsible for initiating, updating and cleanly finishing the behaviors.

TJArk [23] started with the use of their own framework and later transitioned to the framework provided by B-Human also utilizing CABSL for the implementation of their behavior.

rUNSWift [24] team is using Python in order to implement the higher levels of their behavior in favor of faster development and their behaviors were modeled in a decision tree where the node were divided into two categories, roles and skills.

[1] Logic Labeled Finite State Machines.

UPennalizers [25] are using three different state machines in order to implement their behaviors. One is responsible for controlling the game flow and the other two control the head and the body. Their framework is implemented in a combination of Lua and C/C++.

Chapter 4

Our Approach

In this chapter, we are giving a brief overview of our progress, as well as the difficulties we had to overcome. We go over the framework we use, we examine a synopsis of the game states and the most important modules used in our behaviors. Most importantly, we describe in detail the implemented robotic behaviors.

4.1 Initial Steps

Before we started developing behaviors, we had to properly set up the given tools; understand and configure the B-Human code base, our robot and everything else required. This section is a brief overview of the procedure we followed before we reached the point where we were able to focus solely in the behavior implementation.

4.1.1 Code Base 2017

Working with simulation.

Our implementation was initiated while trying to understand the B-Human code base of 2017 [2]. At that time, due to courses and labs we weren't able to test anything on a real robot so most of our experimentation was fiddling with code and testing it on the included simulator, called SimRobot, in order to get a better understanding of the project's hierarchy, since we did not have any prior experience. Through this procedure we were able to figure out how to alternate between behaviors of one simulated robot. To be more precise, we needed to locate the appropriate configuration file in order to

bypass the soccer code that was supposed to be responsible for the role assignment of each robot, as well as the game state transition and replace it with the actual behavior.

Since there was only one behavior included at the time[1], we started experimenting on that behavior testing different options that, based on the documentation, were implemented. At that point, we realized that the majority of the code responsible for the behavior and head control explained in the code release was rightfully removed by the code providers, leaving us with little information regarding the usage of the modules. So, we figured out the basic concepts of CABSL, how to make proper transitions to different states, as well as how to control the movement of the robot with the use of the functions provided with the example behavior.

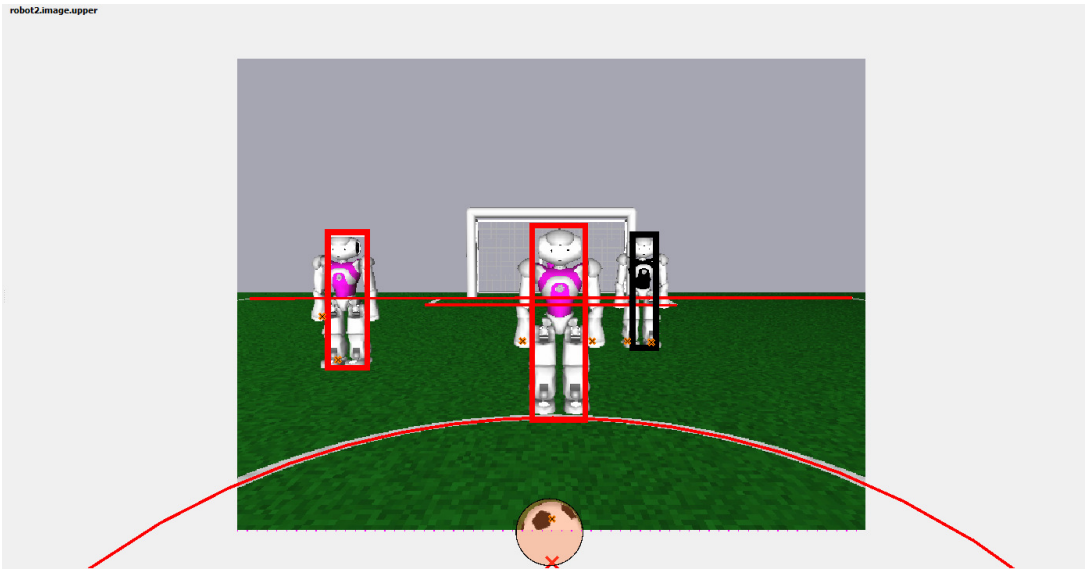
We knew that all the rest of the modules were working properly because we could see the representation in SimRobot views, as shown in Figure 4.1, but we didn't know how to access that information in our behaviors. We couldn't, at that point, understand how to work with obstacles that the robot would identify, but we found how to work with field dimensions, as well as objects with local variables. A few examples were the ball model, as well as the robot's angle with the opponent goal.

Adding a real Nao into the equation.

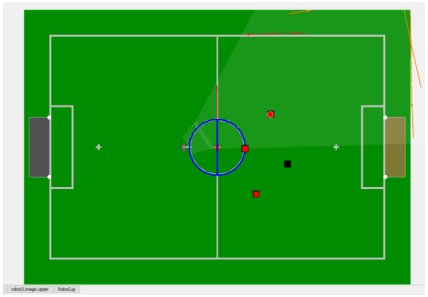
At that time, we were able to begin setting up one of our robots for B-Human code, a Nao v.5 (model of 2014). That required formatting the robot, running the required scripts and setting up the necessary files for networking, as well as a proper setup for the access point. Due to a peculiar setup with static IPs that was required for our older soccer robots to be able to connect to our lab's wireless network, we had to create a new wireless network that would comply with the specifications of the SPL rules [26].

Additionally, we found out how to create different locations (field dimensions, landmark coordinates, a variety of settings, etc.) and switch between them, something very useful for the real robot, since they were also needed for the robot's calibration. Apart from that, any localization would be impossible if wrong dimensions were given. It is important to point out that the field we have in the lab follows the specifications of earlier stages of the SPL 2.3, meaning that its dimensions as well as the goal posts are

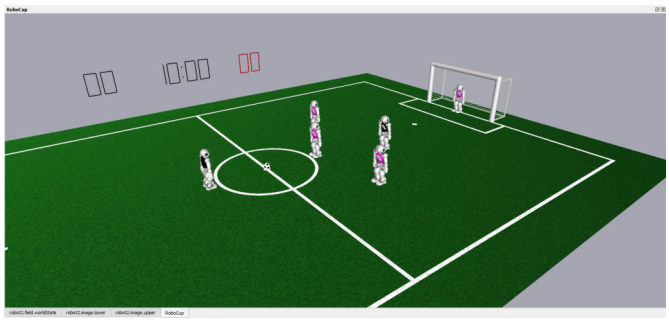
[1] A striker example provided with the project to assist in understanding the code and how to call different modules.



(a)



(b)



(c)

Figure 4.1: In Figure 4.1(a), we can see that the perception module can determine obstacles and also differentiate between different robots based on the color of their jersey. In Figure 4.1(b), we can see the belief the robot has about its location in the world state, shown in Figure 4.1(c), after a localization scan.

different from the ones expected in the B-Human code base. As a result, in order to test a behavior on a real robot from that point on, we have to also make sure we switch the configuration file to the proper location. Having the location properly setup with accurate field dimensions, we proceeded with calibrating Nao's joints using SimRobot and the appropriate scripts provided.

After properly calibrating Nao, we started the implementation of different head control

modes that were responsible for searching and tracking the ball. Thankfully, Nao had no issue tracking the ball in the lighting conditions in our lab nor any other field landmark, so we didn't attempt a camera calibration. In addition, we implemented a few small behaviors executing basic functionality, such as following and kicking the ball around. Since it was the first time we tested our code with a robot, we had to make many adjustments on the speeds given to the motion engines, pointing out yet again one of the differences between simulation and reality. We also determined a way to access the robot's touch sensors, as well as the bumpers. Furthermore, we figured out how to include new behaviors in the project and be able to alternate between them, while bypassing the soccer behavior that was responsible for the robot's game state.

It should be noted that up to this point in the implementation we were using Linux as the operating system hosting SimRobot on a machine that wasn't meeting the requirements, since we ran into more problems when we tried executing the project on more powerful, yet inflexible, computers (different bit-wise OS architecture and hardware not supporting SSSE3 instruction set required for running SimRobot).

4.1.2 Code Base 2018

Shortly after, B-Human code base of 2018 [3] was released with many changes in structure and modules in order to comply with evolving rules, but also a few improvements on perception. We decided that we should go ahead and move all the implementations to the new code base, since it was improved and up to date. Since the changes were only briefly mentioned in the new code release report, we had to puzzle out how to properly migrate the work we had done so far. That led to a lot of confusion, because the changes in the structure made previously-accessible modules inaccessible and a few configuration files were removed. While that might have been done for a greater good and a cleaner project architecture, it made things more complicated for us, so we had to unriddle the structure of the project anew.

Another major problem that was surfaced with the new release was the different OS version that was required in order to run the code base. The code of 2017 was using a Linux Ubuntu 17.04 distribution, while the 2018 code a Linux Ubuntu 18.04 distribution. We attempted to run the new code base to our old Linux distribution, but we were unsuccessful. Without knowing if we would be able to transition properly to the

new code base, we didn't want to risk loosing the progress made so far. So, we setup the new code base to a different, more powerful machine that was running Windows 10, but didn't have access to the robot, keeping the previous configuration unchanged.

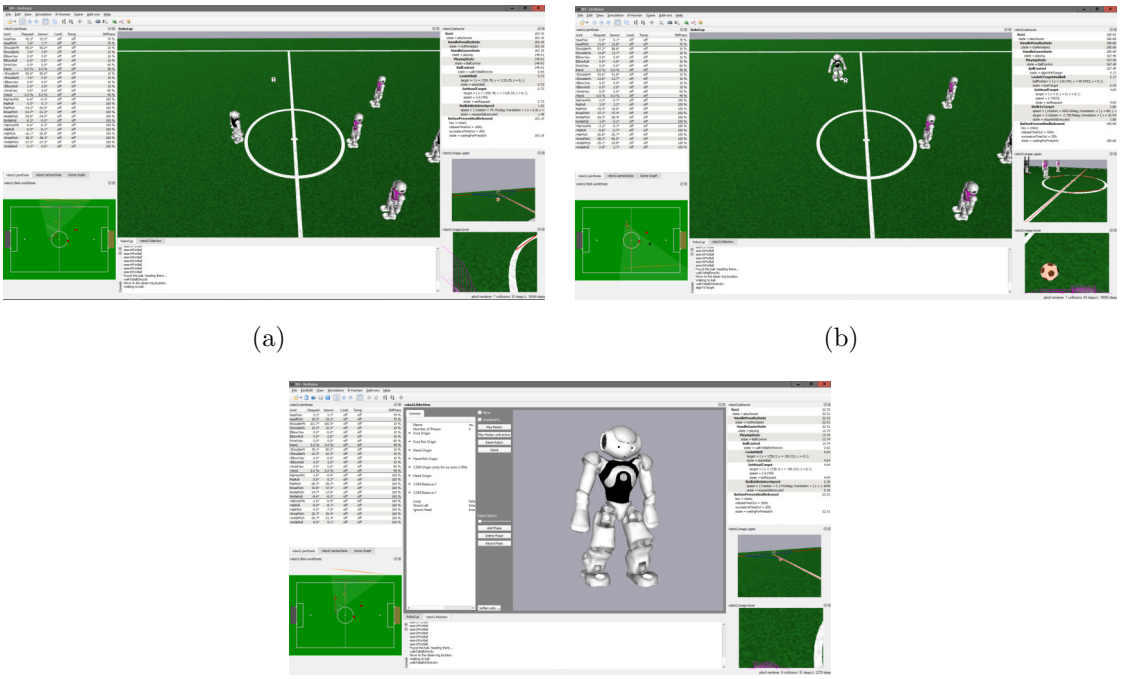
After figuring out how to properly port everything to the new code base and testing that everything functions correctly, we proceeded with the implementation of a few new behaviors in simulation. There was a massive difference in compilation and running speeds due to better hardware, so we didn't proceed into formatting the old machine to the latest distribution.

Moreover, we figured out how to include the executed behavior as part of the playing state, but that created the need for a game controller signal. At that point we became aware of the great functionality of the SimRobot, since we were able to recreate that signal and at the same time monitor almost everything for the simulated robot. In Figure 4.2 we can see all the different views and the information provided for the simulated robot. The same principles also apply to a real robot and there is a brief guide of how to use them in Appendix A. What we didn't realize at that time was that SimRobot is unable to substitute a game controller for a real robot resulting into a few issues when running behaviors on the physical robot. An example issue was that the object with the global coordinates of the ball was get reset constantly.

Understanding how to properly utilize the global and local coordinates of some objects led us to be able to monitor the velocity of the ball and make our robot react accordingly to the ball's, movements based on its position in the field. Also, we understood how to access the information regarding observed objects by the robot and make decisions based on their positions.

Since we were heavily using SimRobot, we needed to control the scenes in simulation, as we did in our lab setup, so we had to work out how to make different scenes to serve that purpose and also find the file that was responsible for their alteration. That also resulted into making some scenes lighter resource-wise.

In the mean time, the computers of our lab were formatted with the required Linux distribution and a few issues appeared with permissions, when transferring files from Windows to Linux via git in order to test our changes on the robot. But with those issues resolved and the computers setup properly, we were able to test each implemented behavior on the robot and make the required adjustments. At that point we came across the game controller issue that resulted in the constant reset of some variables, something



(a) (b)

(c)

Figure 4.2: In Figures 4.2(a), 4.2(b) and 4.2(c) we can see all the different views with information regarding the joints and the world state combined with a brief overview about the state and the movement of Nao. In the top right of each figure there is information regarding the behavior, head control and motion requests. Under it we see the upper, the lower camera views and in Figure 4.2(c) we see in the center view the kick view editor.

that was not occurring in the simulation. A temporary solution, discovered by a team member, was to bypass the handling of the game controller signals, until a way to setup and run a game controller is found.

That was also another point where we realized the difference between the simulated robot and the actual robot and how that difference can damage our robot. Since we were trying to implement more sophisticated movements, we were supplying by mistake the walking engine with a variable that had initially really high values. That resulted into the robot executing sudden movements or even moving at maximum speed for extended periods, something that added stress to its joints. A few noticeable examples were sidestepping at full speed which after a few steps, except from being a bit unstable, was also constantly hitting the leading foot of the motion to the ground. Also, the very

fast overall head movements were distorting the camera stream. So, we started adding weights to some of the variables that controlled the speed of the robot resulting into smoother motion.

Having almost finalized the implemented behaviors, which we describe in Sections 4.7 and 4.8, we started working on understanding how to utilize the different engines in order to perform kicks. Up to this point, we were using the `inWalkEngine` with the default kick provided, but we manage to include different kicks from older code base versions as well. The different `inWalk` kicks that were included from older code releases are better described as dribbles. In addition, we added the code required for generating the request needed in order to perform kicks generated by the kick engine. At first, we used the example kick provided, until we figured out how to use the Kike View, an editor included in SimRobot for creating kicks and other actions with the use of inverse kinematics. We modified the example kick to some extent, but we were unable to proceed further due to constant SimRobot crashes, both on Windows and Linux, when we were trying to add different poses. We still haven't figured out why this happens and we are still unable to run the example kick on our robot due to malfunction of its right shoulder. To be precise, our robot is currently unable to move its right arm and as a result is unable to fully keep its balance in order to execute the kick, resulting into falling onto the ground.

4.2 Game State Flow

Taking for granted that the perception in combination with the cognition have already populated and going to keep the required structures regards the localization and the surroundings of the agent updated, we can solely focus on accessing that information and then make decision based on the world state and the robot's surroundings.

Another thing that should be noted is that in this implementation we bypass the game flow by using the implemented game controller and sending a signal that sets the game state directly to playing for the simulated robot. For the real robot we achieve this by setting our robot manually to penalized state and back. The game flow consists of six states controlled by a module called Game Controller as do the states of the robot, playing state and penalized state.

The game states are being handled by `HandleGameState.h`. In that file the robot basically checks `theGameInfo.state` provided by the game controller and proceeds with

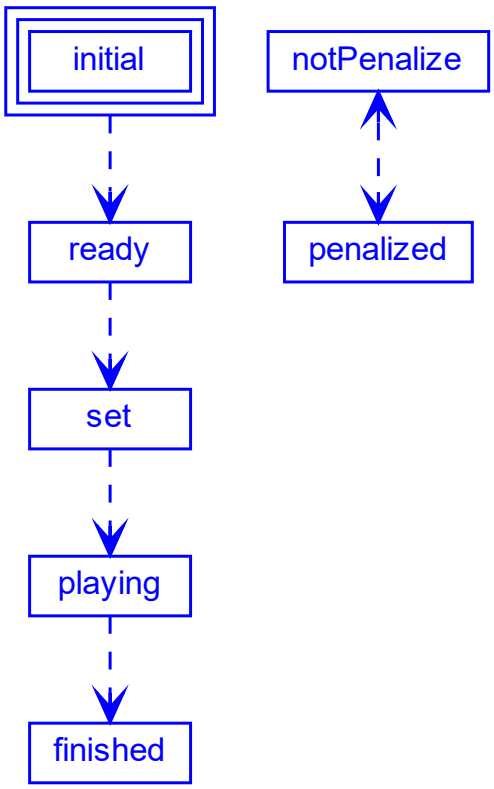


Figure 4.3: Game Controller flow chart.

the execution of the behavior defined for that state. The game states are the following, each one with its own set of rules: initial, ready, set, playing and finished. The robot states are identical with the addition of two more: penalized and getUp. The game flow is represented in Figure 4.3.

initial

This is the initial game state. In this state the head control mode is initialized and the robot is being set to a stand high pose, a natural pose defined as a special action. The

robot is not allowed to do any other kind of movement.

ready

In this state the robots must move to kick off positions. Based on the RoboCup regulations [26], there is a predefined time where the robots must resolve their position and their side of the field and move to their predefined positions based on their role.

set

When this game state is given by the game controller the robots should stand still and wait for the game state to transition to playing or in the latest competition to hear the referee whistle. While they are not allowed to move, they can move their head around or attempt to get up if fallen.

playing

This is where the match starts and the robots play soccer, always complying by the game rules. Robots can be penalized manually in this state by pressing the chest button twice and in quick succession. The behaviors implemented in this thesis are a part solely of the the playing state.

finished

This is the game state that runs at the end of a half and also at the end of the match. It could host cheering or safe shutting down.

getUp

This state is the only state that is not provided by the game controller. Instead, it depends on `theFallenDownState` responsible for handling the case when the robot falls. It is an independent state checked on the higher levels of the behavior, in the same level as the game states, and in the case of a fall the get up motion is decided by a special action.

penalized

When a robot violates the game state rules, it is being set to penalized either by the game controller or manually while pressing the chess button. While in the penalized state it is not allowed to move in any way, neither both body nor head.

Without a game controller running and connected to the network with our soccer robot, we sent the playing state signal via the SimRobot only for the virtual while we were unable to do the real robot. In the simulation, the game controller module functions properly but in the remote control mode, where we are connected to a real robot, we concluded that the connection SimRobot creates with our robot is that of a team mate and not of game controller. As a result, sending any signal regards the playing state has no effect. The only way that we found so far in order to go to the playing after the initial state is to penalize and remove the penalize manually for our robot. Not having a game controller can cause a few more issues that we will examine later on.

Our implemented behaviors are focused only on the playing state and sometimes include actions that should be done on different game states for the sake of simplicity.

4.3 CABSL Behavior Development

In our implementation, the C-based Agent Behavior Specification Language (CABSL) [27] is being used which is the successor of XABSL [28]. This representation has several advantages over its predecessor including a very small coding overhead, support of any datatype used in C++, the ability to perform any kind of C++ computation and also due to its nature, IDEs are able to assist in the code generation. It also has several disadvantages; it allows grammar violations, is limited in C++ and does not directly support on the flight behavior changes although that can be implemented in a different way.

The program that controls a robot is executed in cycles; in each cycle it collects information from its environment and based on the observations makes decisions and acts in order to accomplish its goal.

This approach treats each behavior of a robot or an agent, in general, as a finite state machine, similar to a Mealy machine, called option. Each option is then divided

into states and each state is defined by two sections, transition and action. In the transition section, a transition to another state occurs, if a specific condition defined by the programmer is satisfied. In the action section, a set of commands is being executed, as long as we remain in that state.

In our case, in each cycle, all the reachable options based on the state of the robot are being called and alternate based on the truthful transitions, executing the actions listed on their active states. As a result the implemented behaviors are considered a mapping of the world state in combination with the agent's actions that resulted after a specific sequence of visited states.

For instance, let's assume we are implementing a behavior called `ourBehavior`, as shown in Listing 4.1. This behavior consists of three states. State `firstState` is the initial state, where the execution begins. When we initially call `ourBehavior`, we transition to the `firstState`. In each execution cycle, we remain in that state constantly checking $condition_A$ and executing the actions included in the action section of the state. The transitions are checked before the actions are executed. When the $condition_A$ becomes true, we transition to `secondState` using the `goto` keyword. After transitioning to the `secondState`, we first check in sequence whether or not $condition_B$ or $condition_C$ become true in order to trigger another transition. If neither is true, we execute the action of that state.

Listing 4.1: CABSL example

```
option(ourBehavior)
{
    initial_state(firstState)
    {
        transition
        {
            if (condition_A)
            {
                goto secondState;
            }
        }
        action
        {
            // Do something
        }
    }

    state(secondState)
    {
        transition
        {
            if (condition_B)
            {
                goto firstState;
            }
            if (condition_C)
            {
                goto thirdState;
            }
        }
        action
        {
            // Do something else
        }
    }

    state(thirdState)
    {
        transition
        {
            MySecondBehavior(); // We can also call another behavior.
        }
        action
        {
            // Better not to do something since you are calling another behavior!
        }
    }
}
```

There are other useful symbols define that can be used for conditioning the state transitions such as `option_time, state_time, action_done, action_aborted`.

`option_time` stores the time since the first execution cycle of the option. We don't use this symbol in the developed behaviors.

`state_time` is a timer that stores the duration that the running option stays in the same state after the first execution cycle of the switch. We use this symbol in many different states of the implemented behaviors as we will see in the sections that follow.

`action_done` is a symbol that becomes true if the last sub-option is executed in the previous execution cycle reached a target state. Previously we mentioned that an option is able to call another options. These sub options can inform the option that called them about their state. It used in most motion requests.

`action_aborted` similar with the `action_done` only this time this symbol becomes true if the sub-option reached the aborted state.

4.4 Modules used in Behavior Development

4.4.1 Walking

There are three different ways to control the movement of our robot. Either by setting the speed of the motion with `WalkAtRelativeSpeed` and `WalkAtAbsoluteSpeed`, or by setting a target and also the speed we wish the robot to use in order to reach the given target, `WalkToTarget`. We mostly make use of the first option in our implemented behaviors, while the second is an alteration of the first that uses normal speed values instead of a the percentage of the maximum speed that is used in the first.

WalkAtRelativeSpeed

With this option, a motion request is created with three different speeds, one for the rotation of the robot and another 2 for the x and y axes, that we provide to the motion in the form of a `Pose2f` vector. These values are a percentage of the maximum speed and take values from -1.f to 1.f. As mentioned in the option declaration given, providing the option with `Pose2f(0.f, 1.f, 0.f)` lets move the robot forward at full speed, `Pose2f(0.f, 0.5f, 0.5f)` lets move the robot diagonal at half of the possible speed while `Pose2f(0.5f, 1.f, 0.f)` lets move the robot in a circle. In addition, with

`Pose2f(1.f, 0.f, 0.f)` the robot rotates around itself left, anticlockwise while using a negative value, the robot will rotate right, clockwise. During the `walkAtRelativeSpeed` motion, the robot walks constantly, even when the speed becomes zero. When that happens, the robot keeps walking in place. In order to make the robot stop the walking motion we must call the motion request `Stand()`. The maximum value of all the speeds is 1.0 and from what we understood , even if we set a higher value, the robot is unable to go any faster.

We are able to manipulate the speeds given to this motion request by providing the speed in a manner seen in Listings 4.2 and 4.3. In the first example, by using the angle of the ball relative to the robot as an input for the z axis that controls the rotation of the robot, our robot is able to react to the ball movements. As a result, it constantly tries to turn towards the ball minimizing its angle with the ball. Adding a forward movement (0.8f) to the motion results to a movement where the robot actively chases the ball. For instance, if the ball is located -30 degrees relative to the robot and we pass the -30.f to the rotation ,the robot will start turning right towards the ball(at maximum possible speed). In the same time, that angle is reduced and the updated value is given to the option resulting to the robot walking towards the ball but also chasing it in case the ball changes position. In the second example, we see a more complicated movement in order to walk and align at the given observing position. Instead of breaking it into smaller states, we can do the following. While turning, if the robot's x reaches the value -900.f, the result of the expression `-900.f - theRobotPose.translation.x()` will become zero. As a result, it will keep moving sideways in order to match its y coordinate with that of the ball's global position. In other words until the `theTeamBallModel.position.y() - theRobotPose.translation.y()` become zero. The problem with this approach is that the produced speeds have really high values and need to be adjusted with weights. In addition, it is important to understand that those two motions occur in the same time and we constantly check the robot's coordinates in order to determine if it reached the required location in order to transition to a different state.

Listing 4.2: Walk to ball example.

```
state(walkToBall)
{
    transition
    {
        // Check if the ball has been seen recently. Could also use
            theBehaviorParameters.ballNotSeenTimeOut if properly setup.

        if(theLibCodeRelease.timeSinceBallWasSeen > 3000)
        {
            // If not, we need to relocate it.

            OUTPUT_TEXT("searchForBall");
            goto searchForBall;
        }
    }
    action
    {
        /* A simple movement in order to walk towards the ball. */
        WalkAtRelativeSpeed(Pose2f(theBallModel.estimate.position.angle(), 0.8f,
            0.f));

    }
}
```

Listing 4.3: Walk to observing position example.

```
// The robot walks to the predefind observation point. In this case its his position
    as a defender.

state(walkToObservingPosition)
{
    transition
    {
        // Check if the ball has been seen recently.

        if (theLibCodeRelease.timeSinceBallWasSeen > 3000)
        {
            // If not, we need to relocate it.

            OUTPUT_TEXT("searchForBall");

            goto searchForBall;
        }

        // Check if the ball is on your side of the field where it actually starts
            getting dangerous.

        if (theTeamBallModel.position.x() < 0)
        {
            OUTPUT_TEXT("moveToBallWhileBlocking");
```

```
            goto moveToBallWhileBlocking;
        }

        // Check if the robot reached the observing position.
        // TODO maybe I should add the angle with the ball as well.
        if (theLibCodeRelease.between(theRobotPose.translation.x(), -910.f, -905.f)
            && (theLibCodeRelease.between(theRobotPose.translation.y(),
            theTeamBallModel.position.y() - 5.f,theTeamBallModel.position.y() +
            5.f)))
        {
            OUTPUT_TEXT("standingAndObserving");

            goto standingAndObserving;
        }
    }
    action
    {
        // Call the look at ball with the coordinates of the ball relative to the
            robot.

        LookAtBall(Vector3f(theBallModel.estimate.position.x(),
            theBallModel.estimate.position.y(), 0.f));

        // Also head towards the observing position.

        WalkAtRelativeSpeed(Pose2f(theBallModel.estimate.position.angle(), -900.f -
            theRobotPose.translation.x(), theTeamBallModel.position.y() -
            theRobotPose.translation.y()));
    }
}
```

WalkToTarget

In order to use this option we need to supply the motion engine with two `Pose2f` vectors. The first one is the speed vector while the second vector indicates the coordinates of the destination relative to the robot. We haven't worked extensively with this option so we haven't yet utilized its full potential. We are mostly using this option when we want to handle the alignment of the robot with a target but only under certain conditions, for instance performing a kick. An example can be seen in Listing 4.4 where we see a state that handles the robot alignment with the goal. In this state we also see the selection of the appropriate foot based on the position of the ball in the field. It must be noted that we don't strictly follow the guidelines of CABSL since we are using conditions in the action section. This is a subject for future work.

Listing 4.4: Align with the goal example.

```
// In this state the robot align behind the ball towards the goal.

state(alignWithGoal)
{
    transition
    {
        // Check if the ball has been seen recently.

        if(theLibCodeRelease.timeSinceBallWasSeen > 300)
        {
            // If not, we need to relocate it.

            OUTPUT_TEXT("searchForBall");

            goto searchForBall;
        }

        // If properly aligned kick the ball.
        // The decision about the shooting foot must go a step higher. The foot
            should be decided higher in the hierarchy and then be used here.

        if (((theLibCodeRelease.between(theBallModel.estimate.position.y(), 40.f,
            60.f) &&
             (theRobotPose.translation.y() >= 0.f)) ||
             (theLibCodeRelease.between(theBallModel.estimate.position.y(), -60.f,
                 -40.f) &&
             (theRobotPose.translation.y() < 0.f)))
             &&
             (theBallModel.estimate.position.x() <= 200.f) &&
             (std::abs(theLibCodeRelease.angleToGoal) < 5_deg))
        {
            OUTPUT_TEXT("kickToGoal");

            goto kickToGoal;
        }
    }
    action
    {
        // Call the look at ball with the coordinates of the ball relative to the
            robot.

        LookAtTargetAndBall(Vector3f(theFieldDimensions.xPosOpponentGoal, 0.f, 0.f),
            Vector3f(theBallModel.estimate.position.x(),
            theBallModel.estimate.position.y(), 0.f));

        // Call the look at ball with the coordinates of the ball relative to the
            robot.

        if (theRobotPose.translation.y() >= 0.f)
        {
            WalkToTarget(Pose2f(80.f, 80.f, 80.f),
```

```
                Pose2f(theLibCodeRelease.angleToGoal,
                theBallModel.estimate.position.x() - 200.f,
                theBallModel.estimate.position.y() - 50.f));
        }
        else
        {
            WalkToTarget(Pose2f(80.f, 80.f, 80.f),
                Pose2f(theLibCodeRelease.angleToGoal,
                theBallModel.estimate.position.x() - 200.f,
                theBallModel.estimate.position.y() + 50.f));
        }
    }
}
```

4.4.2 Kicking

Kicking the ball can be achieved either with the use of the Kick Engine, the Walk Engine or even with a special action although there was no work done with the latter.

The principle is the same in all implementations. Each motion has a number of phases, each one with different duration. Each phase can have different values for every joint of the robot. The sequence of the different phases compile a motion. The difference between the three different motions seems to be in the way these values are stored, represented and then used.

Kick Engine

In order to utilize the kick engine, a kick file .kmc needs to be created and stored in the `Config` folder. Creating such a file can be achieved with the use of KikeView integrated in SimRobot. The same can be also done manually but not recommended since there are a lot of variables to consider when creating a motion. More information about motion creation can be found in the appendix. A motion request for the kick engines can be seen in Listing 4.5

Listing 4.5: Forward kick request.

```
/**
 * @param mirrored True or false if we want the kick to be mirror. In a way its a
    choise of the kick foot.
 */
option(ForwardKick, (bool)(false) mirrored)
{
    /** Set the motion request. */

    initial_state(setRequest)
    {
        transition
        {
            // When you are ready.

            if (theMotionInfo.motion == MotionRequest::kick)
            {
                goto requestIsExecuted;
            }
        }
        action
        {
            theMotionRequest.motion = MotionRequest::kick;
            theMotionRequest.kickRequest.kickMotionType = KickRequest::kickForward;
            theMotionRequest.kickRequest.mirror = mirrored;
            theMotionRequest.kickRequest.autoProceed = true;
            theMotionRequest.kickRequest.boost = false;
        }
    }

    /** The motion process has started executing the request. */

    target_state(requestIsExecuted)
    {
        transition
        {
            if (theMotionInfo.motion != MotionRequest::kick)
            {
                goto setRequest;
            }
        }
        action
        {
            theMotionRequest.motion = MotionRequest::kick;
            theMotionRequest.kickRequest.kickMotionType = KickRequest::kickForward;
            theMotionRequest.kickRequest.mirror = mirrored;
            theMotionRequest.kickRequest.autoProceed = true;
            theMotionRequest.kickRequest.boost = false;
        }
    }
}
```

Walk Engine

Another way to perform a kick like motion is via the Walk Engine. In this case, there is no tool that we know of to assist in the generation of a motion so the kick files must be created with a text editor in the form of a .cfg file. Similar files can be found again in the `Config` folder of the project in order to be used as a reference. The resulting kick extends the walking motion as an `inWalkKick` and in order to use it we need to provide the option with a `WalkKickVariant` that includes the type of the kick, the leg that is going to execute the motion and also the starting pose of the kick or in other words the coordinates from where the motion should start.

4.5 Developed Common HeadControl Modes

The `HeadControl` option is implemented as an interface and runs in parallel with the main behavior. As the name states, this option is responsible for controlling the head of the robot by the use of the `HeadControlModes` we implemented using the `LookForward` head mode as a reference that was already included, altering slightly. The implemented `HeadControlModes` are the following:

- LookLeftAndRight
- LookAtBall
- LookAtTarget
- LookAtTargetAndBall

LookForward

This is the simplest of the head control modes so far. It is responsible for pointing the head of the robot forward to a default position. The default position of the robot's head when looking forward is with a pan[1] of 0 and a tilt[2] of 34. That results to a slight bend on the robots head downwards in order to be able to cover with both top and bottom camera its feet and the field in front of the it. This is achieved by using the `SetHeadPanTilt` providing it with the pan and tilt mentioned previously and the speed in degrees of the

[1] Pan is for a side to side motion.
[2] Tilt is for an upwards to downwards motion.

transition we wish the robot to complete. `LookForward` is also used in most of the other head control modes as an initialization state.

LookLeftAndRight

The `LookLeftAndRight` mode was implemented because a scanning motion for the head was needed when the robot is searching for the ball. The motion is divided into three states. The initial state calls the `LookForward` mode and then monitors when the head will stop moving by checking the output of the Head Motion Engine. As soon as the head comes to a stop, we transition to the next state, `lookLeft`. In this state, we set the head's pan while keeping the tilt unchanged with the use of the `SetHeadPanTilt`. In order for the robot to turn his head left we need to use positive value. In this case we use 0.85 with a speed of 70 degrees in order to do a smooth motion and without reaching angles that might stress the head joints. When the head stops moving again, meaning that most probably it reached the targeted position, we transition to the next state, `lookRight`. In this state, we set the head's pan to a negative value, -0.85 in order for the head to rotate to the opposite direction while keep checking the Head Motion Engine and from this point on the mode continues looping between `lookLeft` and `lookRight` until the `HeadControlMode` is changed by another state of the behavior control. The code of this head mode can be seen in Listing 4.6

Listing 4.6: Head mode Look left and right.

```
/** An option intended for scanning purposes in order for the robot to determine
 * several things regarding its surroundings.
 * The time of a full rotation is now unknown.
 * */

option(LookLeftAndRight)
{
    /* Simply sets the necessary angles */

    initial_state(lookForward)
    {
        transition
        {
            if(state_time > 100 && !theHeadMotionEngineOutput.moving)
            {
                goto lookLeft;
            }
```

```
        }
        action
        {
            /* > Pan is from side to side. As a result of this the +0.90 degres is to
                the left of
            * the robot, while -0.90 is to the right.
            * > Tilt is from up and down. This time the + is for the down ward
                movement of the
            * robots head while the - moves the robot head upwards.
            * > The last parameter is the speed that the robot is asked to do the head
                movement.
            */
            //SetHeadPanTilt(0.f, 0.38f ,150_deg);
            LookForward();
        }
    }
    state(lookLeft)
    {
        transition
        {
            if (state_time > 100 && !theHeadMotionEngineOutput.moving)
            {
                goto lookRight;
            }
        }
        action
        {
            SetHeadPanTilt(0.85f, 0.38f ,70_deg);
        }
    }
    state(lookRight)
    {
        transition
        {
            if(state_time > 100 && !theHeadMotionEngineOutput.moving)
            {
                goto lookLeft;
            }
        }
        action
        {
            SetHeadPanTilt(-0.85f, 0.38f ,70_deg);
        }
    }
}
```

LookAtTarget

In this mode, the joints of the head are set to the coordinates of the target relative to the robot, provided to the option as a `Vector3f` target which is a vector containing

three float values, x, y and rotation since it is required from the Head Motion Request `SetHeadTarget`. That request is then passed to the Head Motion Engine that handles the positioning of the head. To transform a global coordinate to a local angle we can use `theRobotPose.inversePose * Vector2f(target.xPos, target.yPos)).angle();`. The code of this head mode can be seen in Listing 4.7

Listing 4.7: Head mode Look at target.

```
/** Turns the head towards the target. The target's coordinates are relative to the
    robot(local).
 * */

option(LookAtTarget, (const Vector3f&)(Vector3f::Zero()) target)
{
    /* Simply sets the necessary angles */

    initial_state(lookForward)
    {
        transition
        {
            if(state_time > 100 && !theHeadMotionEngineOutput.moving)
            {
                goto lookTarget;
            }
        }
        action
        {
            LookForward();
        }
    }

    state(lookTarget)
    {
        transition
        {
        }
        action
        {
              // Local coordinates.

              SetHeadTarget(target, 100_deg);
        }
    }
}
```

LookAtBall

Exactly the same as `LookAtTarget` we use it while providing it with the ball's local coordinates. It was created with a different intention. The main idea was that the behavior wouldn't give the coordinate of the ball to `HeadControl`. Instead, the `HeadContol` should have access to the `BallModel` and get the coordinates by accessing the object of that class but this is not implemented because it needs changes in the project's hierarchy. As the robot moves, the coordinates get updated and as a result the robot tracks the given target.

LookAtTargetAndBall

In the `LookAtTargetAndBall` we pass as arguments two different targets. One for the position of the ball and one more for the target we want to observe, both with their coordinates relative to the robot. The same principle such as `LookLeftAndRight` are followed in this mode. Instead of left and right we cycle through the state responsible for turning the head towards the target, `lookTarget` and `lookBall` that handles the position of the ball.

Additional Head Control Modes

There are a few more head control modes included which try to handle targets on the ground, with the use of their global coordinates, but that motion request while included it does not seem to have the expected results so it has not been utilized so far.

4.6 Developed Common Behavior States

In order to better understand the concept of behavior control as well as the programming environment, four main behaviors were developed:

- Simple Striker
- Simple Defender
- Simple Goalkeeper
- Ball Control

While the first three have a very distinct purpose, Ball Control combines elements from all the rest behaviors trying to implement a soccer-like behavior that wouldn't be directly used in a soccer game. In each behavior, different implementations are being tested in order to fully understand and utilize the language and the tools provided by B-Humans.

From this point onward we need to denote that some states are essential components of the rest of the behaviors and are being used in their integrity or with a minor changes. Those states include:

- Localization scan - `localize`
- Searching the ball - `searchForBall`
- Sidestepping - `patrolLeft/Right`

localize

The initial state of every implemented behavior is the state `localize`. In this state, before the robot starts moving, the robot uses the `HeadControlMode LookLeftAndRight` in order to scan its surroundings and determine its position on the field. This brief state turned out to be really helpful on reducing a bit the unavoidable localization error so it is being used in every other behavior implemented in this thesis. The duration of this state is 8 seconds.

searchForBall

This state is responsible of locating the ball by checking the time since the ball was last seen. That variable it seems to be updated in every frame that the ball is located in the robot's field of view. When the robot sees the ball, the value doesn't go beyond 20 ms. Proceeding with the ball search while the timer has just surpassed 20 ms will result to the robot constantly searching for the ball so we concluded that when the ball has not been seen for more than five seconds, 5000ms allowing room for some movements, is when we need to relocate it. As soon as the ball enters the field of view, the timer is being updated. Checking for a small value of that variable, given that we are in `searchForBall` state, is the condition responsible for transitioning the behavior control to the next state.

patrolLeft/Right

`patrolLeft` is the state where the robot moves sideways based on the ball's velocity in order to follow its path while sticking on a specific trajectory based on the implementing behavior. It also keeps track of the ball velocity and angle with the robot so it can decide the next transition. In order to implement such a movement we provide to the `WalkEngine` values for the angle, x and y coordinates via the `WalkAtRelativeSpeed`.

`patrolRight` is almost identical with `patrolLeft` state. The only difference is the speed value that we provide to `WalkEngine` using `WalkAtRelativeSpeed` that has a negative value for the y axes in order to achieve a sideways motion on the opposite direction.

4.7 Developed Soccer Behaviors

4.7.1 Simple Striker

With the development of the Simple Striker we wanted to experiment with the obstacles provided from the perception as well as the different dribbles. Our striker locates the ball and walks towards it. Checks its position relative to the ball, turns towards the opponent goal and checks whether or not its path is blocked by another robot. If so, it chooses to dribble based on the obstacle position in front of it until it goes close enough to the goal. If its path is clear, it kicks straight to the goal. The behavior consists the following states and its flow chart can be seen in Figure 4.4:

- `localize`
- `searchForBall`
- `walkToBall`
- `alignWithGoal`
- `alignLeftDribble/RightDribble`
- `decideKick`
- `kickToGoal`
- `dribbleLeft/dribbleRight`

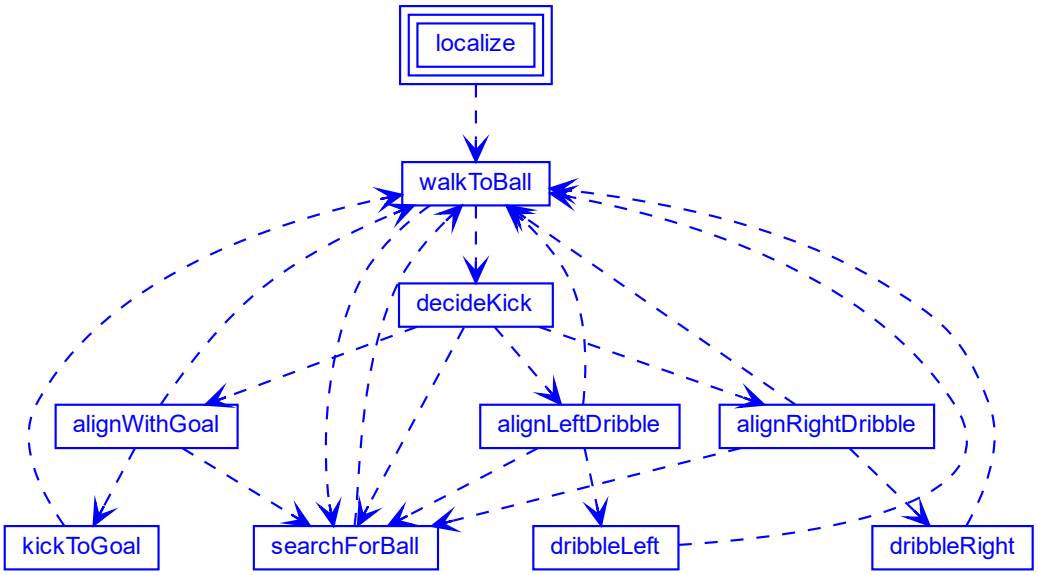


Figure 4.4: Simple Striker flow chart.

walkToBall

In this implementation the `walkToBall` state is a simple ball approach. The robot walks directly to the ball while using its local coordinates and checking if the ball stopped moving and its also close enough in order to switch to the state where the kick is being decided. The head tracks the ball with the use of the `HeadControlMode LookAtBall`.

decideKick

After reaching the ball, the next step is to decide the type of the kick our striker will perform. The type depends on the number of obstacles that are considered opponents and their position on the field based on the robot's field of view while looking at the ball. Obstacles inside the field can be characterized as one of the following:

- `goalpost`
- `unknown`

- `someRobot`
- `opponent`
- `teammate`
- `fallenSomeRobot`
- `fallenOpponent`
- `fallenTeammate`

Regardless of the type, each obstacle includes information about its center, left and right point. In this state we make use of the variable that gives information regards the center of the obstacle. Knowing the dimensions of a Nao, we are able to calculate if the striker can keeps its course without the obstacle interfering and avoiding collisions. If there is an obstacle but is not blocking the robot's path to the goal it proceeds with the alignment with the goal in order to perform a kick it towards it. If that is not the case then the robot checks for an opening. If the y coordinate of the obstacles center is negative or zero it proceeds on align for left dribble and if negative for right dribble. While in this state the robot is not moving executing the motion request `Stand()` and looking at the ball with the use of `LookAtBall.`

alignWithGoal

After deciding to kick towards the opponent's goal, the robot needs to properly align itself behind the ball towards the goal. That movement depends on the angle the robot has when entering this state. If the angle is greater than 60 degrees then the robot needs to do a side ways move, choosing the sorter path, either clockwise or anticlockwise until the angle with the target is between acceptable values. This is achieved with the use of the local coordinates of the ball and moving sideways in order to minimize the robot's angle with the goal. It also checks its position on the field, as seen in Figure 4.5, in order to determine the foot that will execute the kick. Different alignment is required for each foot. If it is located on the upper half of the field the kick will be executed with the left foot so it needs to position itself slightly to the right of the ball. Similarly, for the lower half of the field, the kicking foot is the right so it needs to be position itself more to the left of the ball. It constantly checks its angle with the goal and when that variable reaches an acceptable value the behavior transitions to the `kickToGoal`. Also

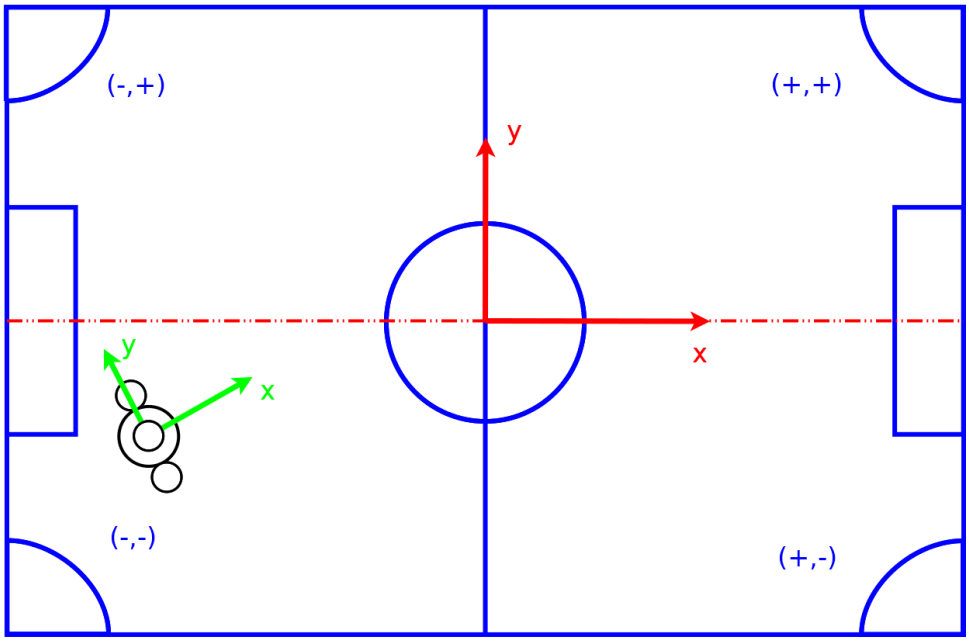


Figure 4.5: On the upper part of the field, where the robot's y coordinate is positive, our striker aligns itself and kicks based on the left foot. The same principle if followed for the right foot in the lower part of the field. Its important to note that its own side of the field is always positioned at the plain where x is negative.

the head control mode moves the head between the ball and the goal position with the use of `HeadControlMode LookAtTargetAndBall`.

kickToGoal

When properly aligned with the goal, the robot proceeds to kick the ball. In this state, while the head control mode `LookAtBall` handles the head motion, the robot proceeds into kicking it towards the goal. That is achieved with the use of an `InWalkKick`, the walk kick variant `forward` with the use of the appropriate kicking leg while also providing to the engine the starting position of the motion. When that motion is finished it triggers the flag `action_done`. Using that flag in combination with a counter that records the amount of time the control is in this state the robot decides whether or not to proceed to the next state. This is useful in case something goes wrong the kick and is never executed correctly since it will be able to continue with the execution of the behavior.

alignLeftDribble/alignRightDribble

Since there is an obstacle blocking the robot's path to the goal our striker needs to avoid it. It needs to position itself in such a way in order to align for the appropriate foot for each dribble. Since we want to dribble towards the goal, the angle with the goal is again being used in order to perform the alignment but with a 10 degrees offset. The `inWalkKicks` that will follow these alignment maneuvers instead of a straight forward kick, turn slightly the torso of the robot towards the orientation of the kick so it needs to be positioned a bit differently. Also it needs to be noted that `alignLeftDribble` transitions to `dribbleLeft`, while `alignRightDribble` to `dribbleRight` but the kicking leg is the opposite since the robot is performing a dribble towards the sides.

dribbleLeft/dribbleRight

In order to perform a dribble we are sing the `inWalkKicks` with the `turnOut WalkKick` parameter as well as the appropriate leg, left for right dribble and right for left dribble. Each dribble has a different starting pose. Also the resulting kick is a movement with a slight longer execution time. So, in order for the dribble to execute fully we need to add am additional of 3 ms to the execution time of the state before transitioning back to `walkToBall`. During the dribbles we use the `HeadControlMode LookAtTargetAndBall` for extra accuracy.

4.7.2 Simple Defender

The next behavior is the simple defender where we tried out different approaches of the patrol movement. The defender's area of action is consider to be below the half line since it is responsible of protecting its own goal. As results, this behavior constantly checks the position of the ball and re-positions the robot close to it without passing the field line. In addition, if the ball reaches its own part of the field, it proceeds on kicking it away. This behavior consists the following states and its flow chart can be seen in Figure 4.7:

- `localize`
- `searchForBall`
- `decideNextMove`
- `moveToBallWhileBlocking`

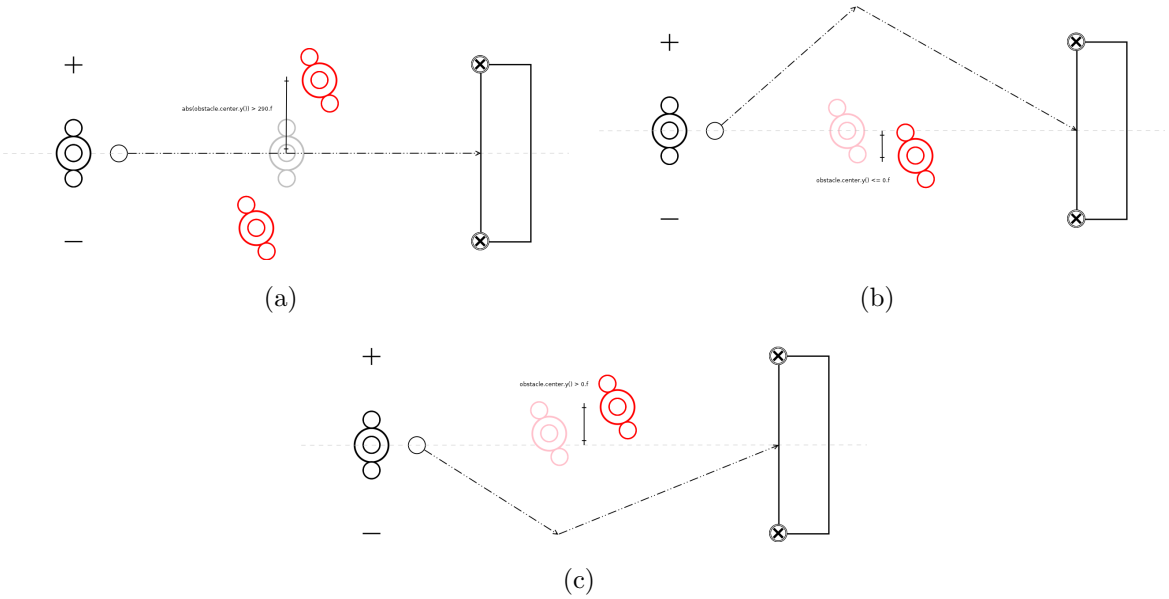


Figure 4.6: In Figure 4.6(a) we can see when there is no need for a dribble while on Figures 4.6(b) and 4.6(c) we see when a left and when a right dribble is being chosen.

- `alignForKick`
- `kickAwayFromGoal`
- `walkToObservingPosition`
- `standingAndObserving`
- `patrolLeft/patrolRight`

decideNextMove

After the state `localize` that is responsible for localization, our behavior control transitions to the state `decideNextMove`. In this state, the defender checks the global position of the ball in order to act accordingly. If the ball is located at its own part of the field, then the defender proceeds into walking towards the ball in order to kick it away and so the behavior control transitions to `moveToBallWhileBlocking`. If the ball is on the opponent's half of the field it then heads to the observing position with the state `walkToObservingPosition`. During this state the robot stands still with the use of motion request `Stand` and tracks the ball with `LookAtBall`.

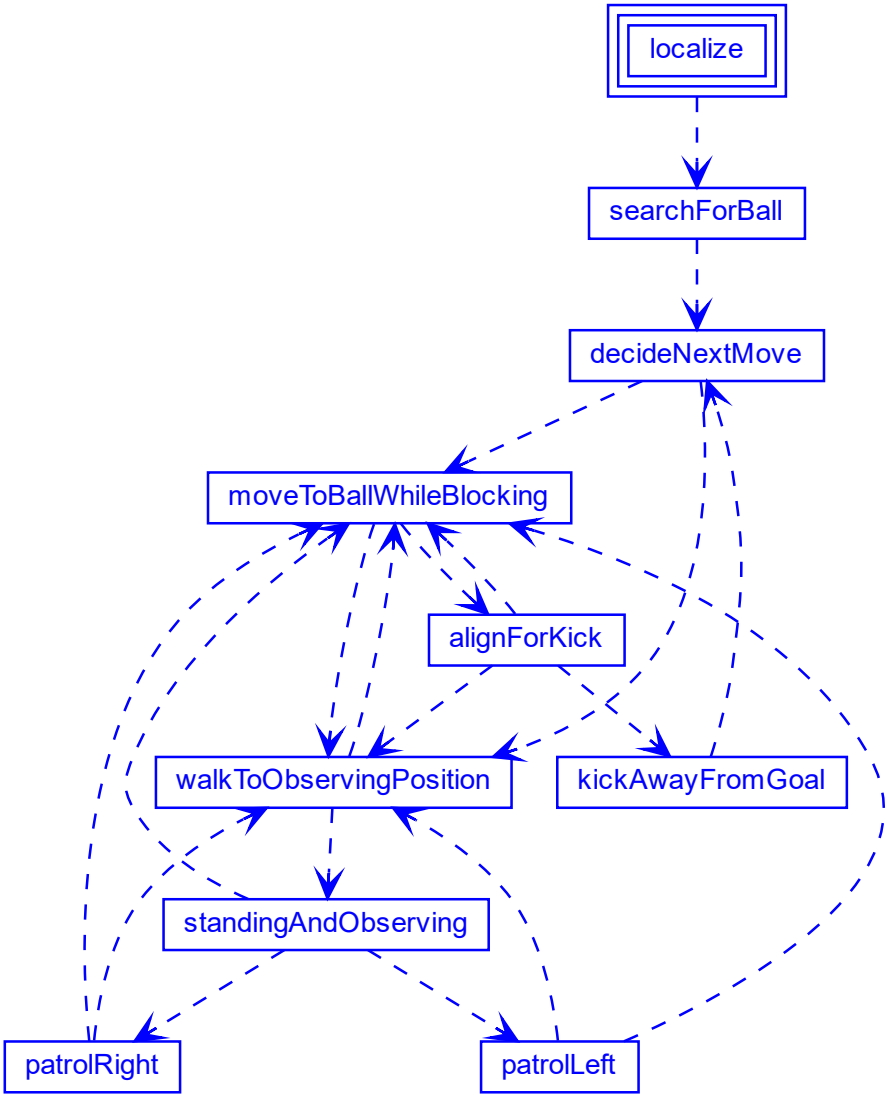


Figure 4.7: Simple Defender flow chart. All of the states except `kickAwayFromGoal` lead to `searchForBall` but the edges are removed from the chart for the sake of clarity.

moveToBallWhileBlocking

In this state, our defender walks straight to the ball while checking if by any chance it passed the center line again, something that would also trigger a transition to the behavior control. In that case, `walkToObservingPosition` is the state that would be called. If that does not happen, the robot keeps moving towards the ball until it reaches the required distance in order to transition to `alignForKick` that is responsible for the realignment of the robot before the kick. While moving towards the ball our defender tracks it with the `LookAtBall`.

alignForKick

At this point, our robot is at a respectable distance from the ball and starts walking side ways in order to properly align with the ball and the angle that is needed for the kick to the ball away toward the opponent goal. The head keeps alternating between the ball and the opponent goal with the use of `LookAtTargetAndBall` while checking if the ball has been moved further away. If so, it will stop aligning and try to move closer to the ball with the state `moveToBallWhileBlocking` and if it passed the center line it will transition to `walkToObservingPosition`. It also checks the angle it has with the ending target.

The alignment move is also divided into two parts although is important to note that we are now referring to the movements the robot executes after arriving to the ball. The first part is responsible for the movement required if the target requires a turn bigger that 60 degrees while the second part handles the movement pass that point. Also, it takes into consideration and chooses the smaller angle as well as the proper foot for a smooth kick. When the required angle and proper distance has been reached it transitions to `kickAwayFromGoal`.

kickAwayFromGoal

Having properly align with the target and the ball the robot proceeds to kick the ball away from its side of the field and its goal while looking at the kicking target with the use of `LookAtTarget`. The kicking foot depends on the position of the robot on the field as seen in the striker behavior. After the kick has been executed or a specific time has

past the behavior control transitions back to `decideNextMove` in order to conclude on the next move.

walkToObservingPosition

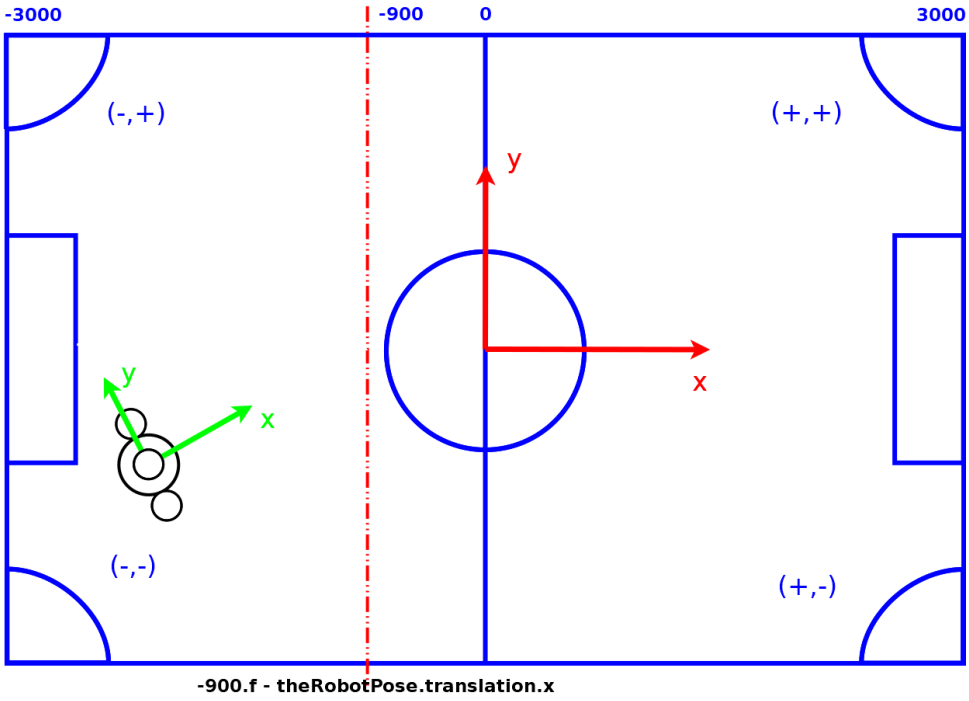
If the ball is located on the opponents side of the field the defender walks to the observing position. This position is located on a line a bit behind of the center line as shown in the Figure 4.8, where x = -910 while y depends on the global coordinate y of the ball. That results into an observing position that changes based on the ball position. In addition, since the value for the sideways motion can be around 1.f most of the time due to the nature of the calculation we limit the motion speed as follows. If the absolute value of the resulting speed is way beyond 0.85f we set it to 0.85f and using the fraction of the speed divided by its absolute value we extract the sign of the motion. While in this state the robot keeps tracking the ball with `LookAtBall` through out the movement and also checks whether or not it reached the observing position in order to transition to the state `standingAndObserving`.

standingAndObserving

After reaching the observing position, our defender stands still and passively tracks the ball with `LookAtBall` as long as its velocity is zero. As soon as the ball starts moving, its velocity becomes non zero and based on its sign it will either transition to `patrolLeft` if its positive or `patrolRight` if its negative. It also checks if it pass to its side of the field in order to transition to `moveToBallWhileBlocking`.

patrolLeft/patrolRight

While being in the `standingAndObserving` state if the ball starts moving, the defender executes a side way movement depending on the movement of the ball. In the same manner, tracking the ball's velocity provides to the defender information regards its movement. So, it constantly checks the local velocity of the ball while tracking the ball with `LookAtBall` and in the same time being ready to transition to `moveToBallWhileBlocking`. Also, it seems that when the robot looses track of the ball then its velocity relative to the robot also becomes zero. As a result, we can use this state as a correction state for small movements of the ball but also a brief state before `walkToObservingPosition`.



(a)

Figure 4.8: Simple Defender positioning.

4.7.3 Simple Goalkeeper

The last soccer behavior implemented is a simple goalkeeper. His objective is to protect its own goal by blocking its path towards its goal and kicking the ball away when it enters its penalty area. It also assists its team by scanning the field and providing them with useful information regards their location, the opponents location as well as the ball's location. For that to be achieved it should be able to identify the goal's location, to position itself there and maneuver accordingly. The states for this behavior are the following and the behavior's flow chart can be seen in Figure 4.9:

- `localize`
- `searchForBall`
- `searchForBallWithMovement`
- `turnToGoal`
- `walkToGoal`

- `alingInfrontOfGoal`
- `turnToCenter`
- `scanField`
- `trackTheBall`
- `activeTracking`
- `alignForKick`
- `kickAwayFromGoal`
- `blockTheOpenCorner`
- `trackTheBallWhileBlockingTheCorner`

An interesting thing to note is that during the localization state, `localize`, it is not required for the goalkeeper to see the goal posts, it can conclude on their location based on other landmarks since the field dimensions are known and consider to be ground truth.

searchForBall

This state is the same used on the rest of the behaviors with only a couple of differences. Firstly, the goalkeeper does not revolve around itself when searching the ball since the ball should be located inside the field. Additionally, if the behavior control stays more than 20 seconds in this state it could mean that there is an issue with the localization, thus it transitions to `searchForBallWithMovement`. If the ball is located, then it transitions to `activeTracking`.

searchForBallWithMovement

Since there is not any other robot to assist with the localization, the goalkeeper might get stuck on a dead lock when reaching its goal. This occurs because there are no landmarks past that point so it can easily misapprehend its position, specially its angle. So, we concluded that if it takes way too long for our goalkeeper to locate the ball it should act in order to find it. In this state, the goalkeeper continues its search for the ball while turning around itself as the rest of the behaviors do. This way, it is also able to spot field landmarks such as the penalty line and the penalty mark. The presence of such landmarks is able to resolve any issues in localization, allowing the behavior control to proceed normally. As soon as the ball is located, it transitions to `activeTracking` as the normal `searchForBall` does.

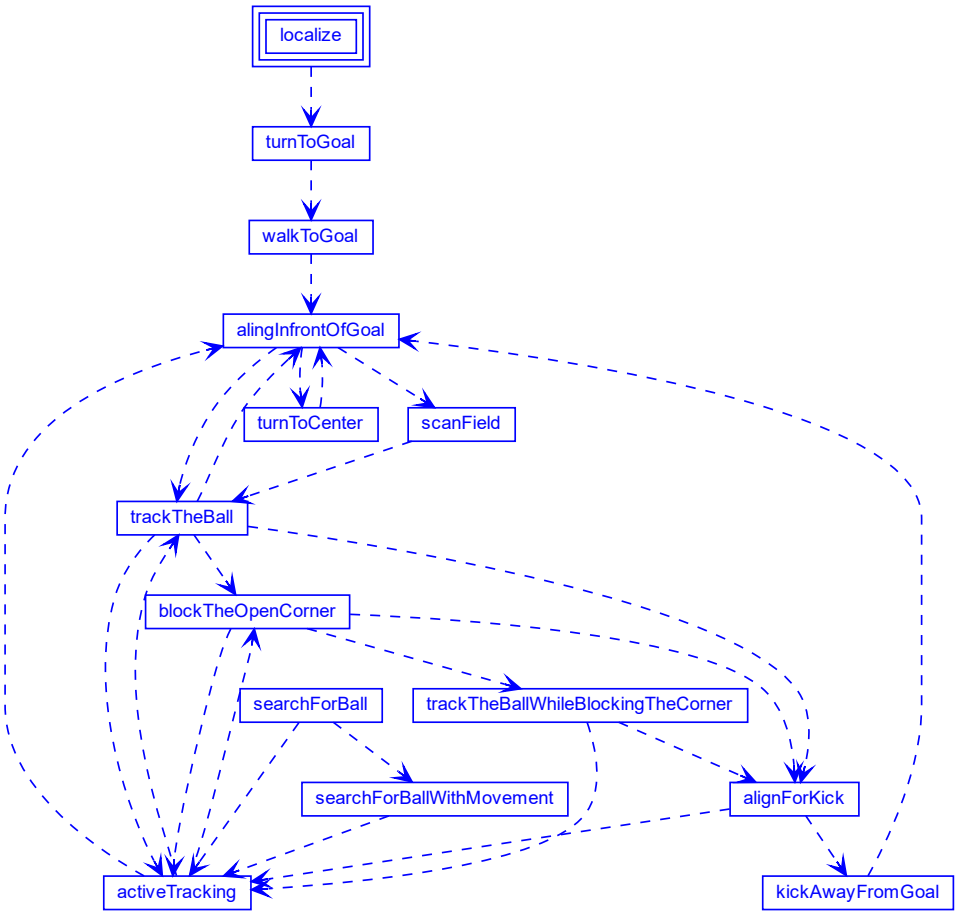


Figure 4.9: Simple Goalkeeper flow chart. The states `scanField`, `trackTheBall`, `alignForKick`, `blockTheOpenCorner`, `trackTheBallWhileBlockingTheCorner` and `activeTracking` also lead to `searchForBall` but the edges are not displayed on the graph for clarity.

turnToGoal

After having concluded on its position on the field, the next step is to turn towards its own goal. That is achieved with the use of the `fieldDimentions xPosOwnGroundline` as well as the global robot pose. Calculating the angle between the vector that describes

the pose of the robot and the goal ground line `(xPosOwnGroundline,0)` while revolving around its own axis will align the robot with its own goal. In the same time, it checks constantly whether or not it reached an acceptable angle so it can be considered facing its own goal and then transition to the state `walkToGoal`. Its head looks forward during the execution of this state with the use of the `headControlMode LookForward`.

walkToGoal

A pretty simple state where the robot walks towards its own goal. The idea is the same with the walk to ball with the only difference being the values provided to the walking engine via the `WalkAtRelativeSpeed`. Checking constantly the distance from the `xPosOwnGroundline`, it transitions to the `alingInfrontOfGoal` as soon as the distance requirement is being met. While in this state the robot keeps looking at the goal with the use of `LootAtTarget` mode, providing it with the coordinates of its goal.

alingInfrontOfGoal

The state `alingInfrontOfGoal` is responsible for keeping the robot properly centered in front of the its goal but also acts as connecting state to the states that follow. Its being visited often in order to decide which state to transition to after. In addition, this state in combination with `turtToCenter` is responsible for the maneuver that aligns and centers the robot in front of the goal. Since the angle between the pose on arrival and the desired pose is obtuse, is expected to create an offset to the final pose so this state corrects it. Since the state is being executed when the robot is either facing the goal or the center of the field, it checks the angle of the robot in order to determine which case it is and acts accordingly. Also, in this state the robot checks the position of the ball. If it has passed the center line and it is located on its team half of the field, then the behavior transitions to `trackTheBall` in order to track it. If it is not, then it is not considered as a threat and the robot proceeds to the `scanField` in order to be able to provide its team with information. During the execution of this state, the goalkeeper looks left and right with the use of `LookLeftAndRight` that also assists the localization.

turnToCenter

This state is only used after `alingInfrontOfGoal` in order to execute a turn towards the center of the field. This is achieved with the use of `WalkAtRelativeSpeed` and providing only the robot's angle with the center of the field making the robot revolve around its axis while constantly checking that angle. When it reaches the desired angle it transitions back to `alingInfrontOfGoal` in order to keep the robot centered. It also checks and executes the shortest turn available. During this state, the robot looks left and right with the use of `LookLeftAndRight`.

scanField

In this state, the goalkeeper scans the field using the `HeadControlMode LookLeftAndRight` in order to keep the perception of the field and the other robots updated and accurate. It stays in this state as long as the ball is not considered a threat. As soon as the ball becomes a threat, it transitions to actively tracking the ball with `trackTheBall`.

trackTheBall

When the ball enters its own side of the field, is considered to be a potential threat for the goalkeeper's goal. As a result, the goalkeeper starts tracking it with `HeadControlMode LookAtBall`. From here, based on the ball's position and velocity, the behavior control will transition to the appropriate state. It will transition back to `alingInfrontOfGoal` if the ball moves to the opposite side of the field. If global y coordinate of the ball is between the y coordinates of the left and right goal post as seen in Figure 4.10 in the blue area, then it proceeds to actively track the ball with state `activeTracking` aiming to cover the goal from a direct kick. If the global y coordinate of the ball is not located between the y coordinates of the left and right goal post, as seen in the same figure in the yellow area, it proceeds to block the appropriate corner of its goal with the `blockTheOpenCorner`. Lastly, if the ball enters its penalty area, the green area, it proceeds into aligning with it in order to kick it away with `alignForKick`. During this state it keeps track of the ball with `LookAtBall`.

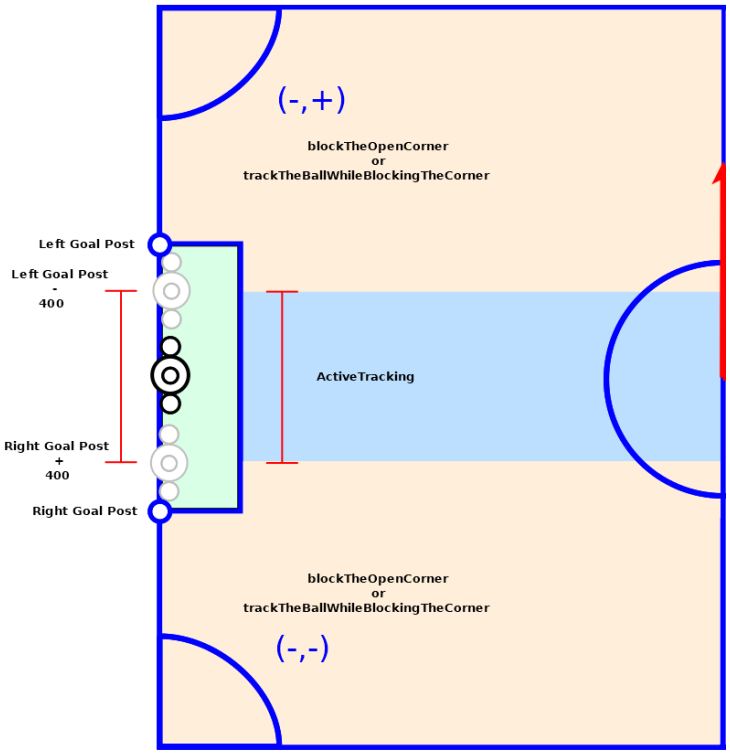


Figure 4.10: Simple Goalkeeper different action areas.

activeTracking

The purpose of this state is to protect the goal from a direct kick. During this state, the goalkeeper tries to match its y coordinate with the y coordinate of the global position of ball as long as it is between the two goal posts minus a static value of 400 in order for the robot to avoid any collision with the posts. As soon as the ball passes those two points, the behavior control transitions to `blockTheOpenCorner`, specially if the robot has not manage to catch up to the ball's position thus far. Similarly with the `trackTheBall` it will transition to `alingInfrontOfGoal` if the ball moves to the opposite side of the field. It also checks if the ball has entered the penalty area in order to proceed with the alignment for kicking it away via `alignForKick`. During this state it keeps track of the ball with `LookAtBall`.

alignForKick/kickAwayFromGoal

`alignForKick` and `kickAwayFromGoal` are the same states as seeing in the Simple Defender. If the ball is moved outside the penalty area it transitions to `activeTracking`.

blockTheOpenCorner

In this state the ball is located outside of the active area of the goalkeeper, either above the left goal minus 400 or below the right goal post plus 400. Is important to note that the y coordinate of right goal is the exact opposite of the left goal. Since the ball is outside the active area, the goalkeeper must move in order to block the opening created in the corner closer to the ball. During this state, the robot's head alternates between the ball and the penalty mark for extra localization accuracy with the use of `LookAtTargetAndBall`. As soon as the robot reaches the relative position it then transitions to `trackTheBallWhileBlockingTheCorner`. If the ball moves back inside the goalkeeper's active area it transition to `activeTracking`. Similarly with the active tracking it also checks if it is required to kick the ball outside of its penalty area.

trackTheBallWhileBlockingTheCorner

After reaching the predefined blocking position, the robot is now blocking the open angle. From this point on, it tracks the ball with `LookAtBall` but without moving. If the ball enters the penalty area it aligns with it in order to kick it away via the `alignForKick`. If the ball moves back inside the goalkeepers active area then the it will start actively tracking it with `activeTracking`.

4.8 Developed Ball Control Behavior

The idea behind this behavior was to combine the functionality of the rest of the behaviors but also try different approaches for walking to ball, aligning and patrolling. That is achieved by having as an objective to keep the ball inside the center circle. That includes searching for ball, walking to ball directly or with an arc, active and stationary ball tracking and also side walking based on the ball's velocity. This behavior is composed by the following states and its flow chart can been seen in Figure 4.11:

- `searchForBall`

Walking to the ball is handled by two states:

- `walkToBallDirectly`
- `walkToBallWithAnArc`

States that are required for kicking:

- `alignWithTarget`
- `kick`

States revolving around the observing position:

- `walkToObservingPosition`
- `standingAndObserving`
- `patrolLeft/patrolRight`

searchForBall

This specific behavior is special since the transition to the state `searchForBall` is a global transition meaning that is being checked in every single state transition regardless of the previous state. This has a couple of benefits. When making the transition global it doesn't need to be included in every single state but is always being checked when transitioning between the states. It might also be slightly faster because we never enter a state in order to transition to `searchForBall` straight after. The same principle applies in this transition as well, we check the time since the ball was last seen and if any robot has not seen the ball for three seconds, 5000ms, it needs to be relocated it. But there is also a negative, if not properly conditioned it can skip the initial localization phase.

4.8.1 Walk to the Ball

The first state that the behavior control transitions to after the state `localize` is the `walkToBallDirectly`. In order for the robot to start moving towards the ball after locating it, a combination of local as well as global variables are used. At first, the robot start heading toward the ball using its local coordinates. In that representation the ball's coordinates are relative to the robot. When it has properly aligned with the ball, checks whether or not at the moment of its arrival to the ball's location the target, in this case the circle in the center of the field, is located at an angle greater than 90 degrees. If that is not the case then the robot moves directly to the ball by executing

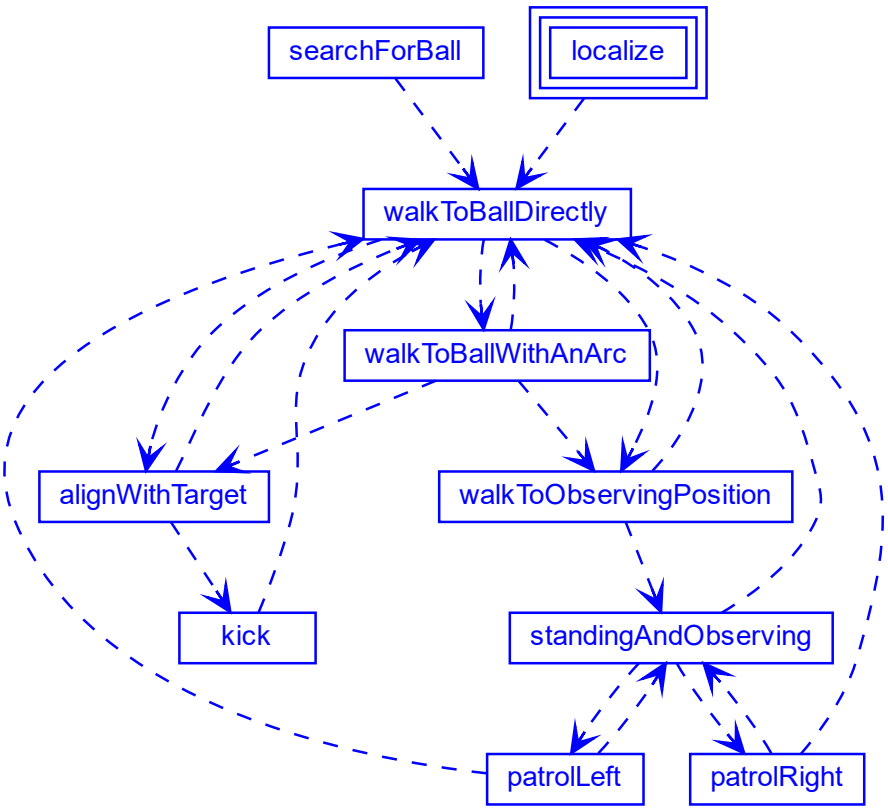


Figure 4.11: Ball Control flow chart.

the `walkToBallDirectly` state. On the contrary, if the greater than 90 degrees, it then chooses a different approach called `walkToBallWithAnArc` state. In both states, the robot's head keeps track of the ball with the use of the `HeadControlMode LookAtBall`. The movement of the robot's body is being controlled with the `WalkAtRelativeSpeed` where we provide the values of the three axes that we saw at section 4.4.1.

walkToBallDirectly

While being in this state, the robot continuously checks a variety of things. It makes sure that the ball is still not located at the target or targeted area. If the ball is moved inside the targeted, it transitions to `walkToObservingPosition`. It also checks whether or not it should keep walking to ball directly or if it needs to transition to another state. Lastly, it must check whether or not it reached an appropriate distance from the ball in order start aligning with the target. If the distance from the target is the appropriate it transitions to `alignWithTarget.` During this state the robot keeps track of the ball with the `LookAtBall.` The robot's approach when walking to ball directly can be seen in Figure 4.12.

walkToBallWithAnArc

In this state, since the robot acknowledges that its angle with the target at the point of arrival to the target is greater than 90 degrees, it moves to the ball while having a slight arc. The robot's approach when walking with an arc towards the ball can be seen in Figure 4.13. That is achieved by multiplying x, y with specific, distance dependent, weights in order to give to the robot a helical trajectory towards the ball while keeping the speed in x axis two times faster than y axis. Arriving at the ball's location with an angle relative to the target, makes the following movements of the robot more smooth and the whole procedure faster. It is important to notice that if the ball is moved and the angle changes the approach changes accordingly. So, in the same manner as `walkToBallDirectly`, if the distance from the target is the appropriate it transitions to `alignWithTarget` and throughout this state the robot keeps track of the ball with the `LookAtBall.`

4.8.2 Kicking the Ball

alignWithTarget

When the robot is close enough to the ball but also in a safe distance in order to avoid pushing the ball around, it executes the actions of `alighWithTarget` state. The goal of this state is to turn the robot behind the ball in order to face the target and then proceed for a possible kick. Similarly with the rest of the behaviors when aligning with

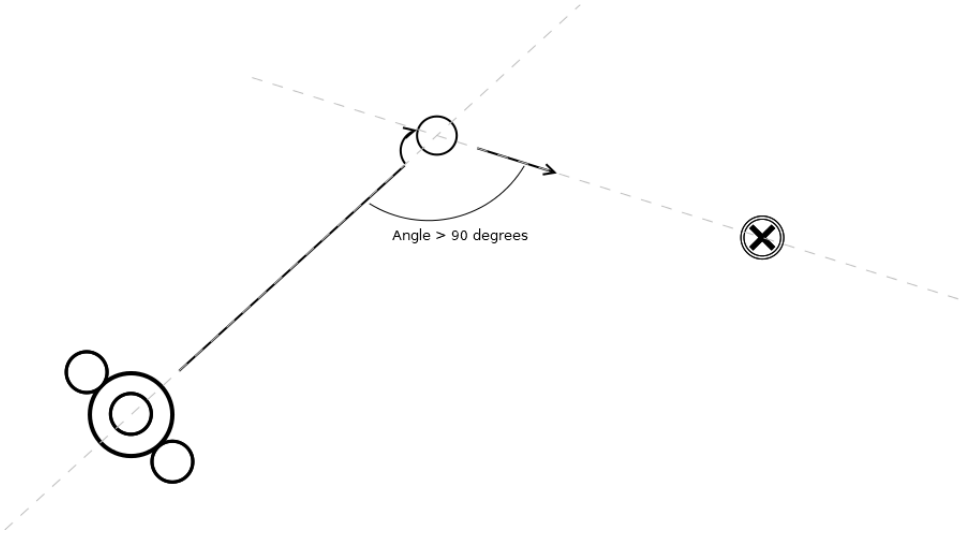


Figure 4.12: Ball approach directly.

their target, the movement depends on the angle the robot has when entering this state and is also divided into two parts although is important to that we are now referring to the movements the robot executes after arriving to the ball. If the angle is greater than 60 degrees then the robot needs to do a side ways move, choosing the shortest path, either clockwise or anticlockwise until the angle with the target is between acceptable values. While in this state, the robot's head cycles between the target and the ball with the use of head control mode `LookAtTargetAndBall`. As soon as the required angle is being met, the alignment maneuver is finished and the control transitions to `ballControlKick`.

kick

Kicking the ball follows the same principal with the rest of the behaviors, thorough explained at section 4.7.1, with the only differences being the target of the motion and a static kick leg, the left leg. In this state, the given target is the center of field.

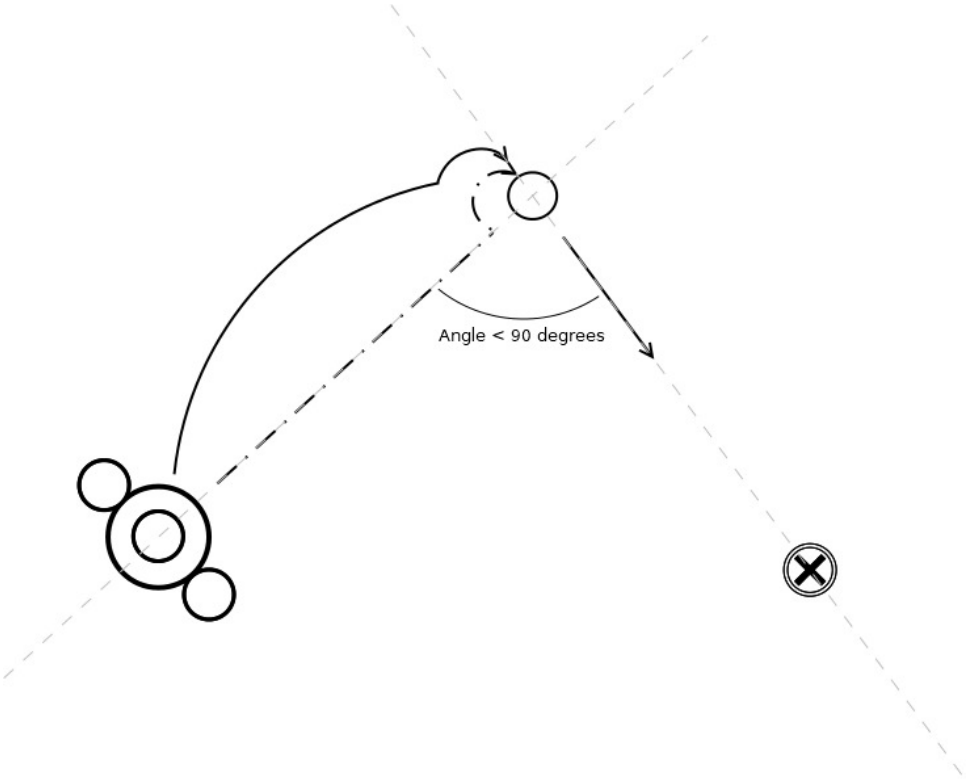


Figure 4.13: Ball approach with an arc.

4.8.3 Observing Position and Actions

walkToObservingPosition

When the ball is located on the target or in this case within the target area, the robot proceeds with moving to the observing position. The head keeps looking at the ball with the `LookAtBall` in order for the robot to keep validating its position and the robot moves with the use of `walkInRelativeSpeed` in order to reach the observing position. We set the speeds of the `walkInRelativeSpeed` with first variable being the angle between the target and the center of the circle and the second being described in `patrolLeft/Right` state in equation 4.1 while keeping the third to zero.

standingAndObserving

In this state the robot stands still while tracking the balls movement with its head using `LookAtBall`. If the ball's velocity changes in the y axes of the local coordinates it then proceeds in patrolling left with the `patrolLeft` state or right with the `patrolRight` state, based on the sign of y.

patrolLeft/patrolRight

As mentioned earlier, `patrolLeft` is the state where the robot moves sideways based on the ball's velocity in order to follow its path while sticking on a specific trajectory, in this behavior that of the center circle line. It transitions to `standingAndObserving` state when the ball has stopped moving and when the angle of the robot with the ball is zero. The angle provided is the angle between the current position of the robot and the center of the circle, calculated with the function `angleTo` from `Geometry.h`. It calculates the angle between a vector, our robot's coordinates and a pose, the center of field using atan2, a two argument `arctagent`. The x coordinate given is calculated as following while y is kept static at 0.85f:

$$x \ : \ \textit{the robot's x coordinate.}$$

$$y \ : \ \textit{the robot's y coordinate.}$$

Since we are interested in following a circle line we can check if we are a radius distance away from the center of the circle. It also important to note that this approach is viable since the center of that circle is also the point of reference for these coordinates.
So:

$$a = x^2 + y^2$$

$$b = (r + 160)^2$$

b is the wanted distance we want our robot to keep from the center.

$$(a - b)/100.000 \tag{4.1}$$

Subtracting these two and then divide them with 100.000 in order to normalize the result to usable numbers. The result will indicate to the robot to move either forward,

or backwards in order for the result to converge to zero thus closely following the circle. Both the static variables 160 and 100.000 were chosen empirically in the simulation environment and seem to behave well in the lab. Especially the 100.000 is needed to normalize the whole result in such a way that the value given to x will not overtake the value given on the y axis of the robot because we want the main movement to be sideways and not forward or backwards. On the y axis we give a value of 0.85, with the max being 1.0. While in this state the robot's head is turned towards the ball, using the `LookAtBall`, constantly check if the ball in contained in the specified area. If not, the behavior transitions back to the `walkToBallDirectly`. The same principle is followed in `patrolRight` with a negative value.

Chapter 5

Results

We implemented the three basic soccer behaviors required for a robotic team to play soccer and since these behavior are distinct they could be viable with the use of a static team strategy. In addition, the last behavior implemented is capable of keeping the ball in the targeted area and also retrieving it in case it is removed.

These behaviors can be executed in simulation as well as in a real environment although some kicks don't seem to be as executable on a real robot due to hardware wear. An easy noticeable example is the malfunctioning joint on the right shoulder than cause a few issues with the balance since the right arm is immobilized. Hardware difficulties indicate the importance of a good simulation tool, hence without SimRobot we wouldn't be able to understand the code or implement anything.

So we can conclude that we are able to properly control our robot with the use of this framework and this representation language without being limited only to soccer behaviors. Also the variety of the tools provided with the framework as well as their modularity makes experimenting and changing implementation on different parts of the project relative easy as soon as the developer grasps the project's architecture.

In this chapter we will go over few scenarios of the implemented behaviors. Each scenario is going to be followed with images of its execution in both simulation and reality. In the following images is important to notice the feedback given by SimRobot in both mentioned cases. The ball is represented as an orange circle. The obstacles are represented in squares with different colors, red for the opponent players and white for the rest obstacle types. In addition, we can see information regarding the touch sensors of the robot represented with squares of magenta color as well as the trigger of the

bumpers while kicking the ball represented with circles. We can see the camera feedback from both cameras of the Nao in the bottom right. Above those two views, we have a camera view from a different perspective that was recorded during the execution of the behavior and added afterwards during the video rendering. In the center of the screen-shots we can see the world state and the belief the robot has about its state and position on the field. Its important to mention that the field has a static representation. That means that our own side of the field is always located on the left (blue area) of the field representation while the opponent's side on the right (red area). That is translated to the negative values on the x axis for our own side and the positive for the opponent's. The robots conclude on which side its their own based on the positions they have at the beginning of their executed behavior. Also we can see a representation (red vectors) of the robots motion and the ball's velocity as well as an anticipation of the ball's position after the kick, small circle in purple color. On the bottom left we can see the console that can provide us with feedback regarding the behavior execution. Above there is a view where we can see the different layers of the executed options and behavior. Both simulation results and lab results follow the principle idea. The only difference is that in the simulation the world state is located under the camera views in the bottom right. The screen-shots of the simulation are capture from a captured video on a machine that runs Windows 10. The screen-shots from the lab are captured from a rendered video that combines a desktop capture of a Linux machine that runs SimRobot and is connected to our soccer player via WiFi in combination with the actual footage of the behaviors recorded from a phone camera. You can find the code in our GitHub repository and the videos can be found at our team's Youtube channel TeamKouretes.

Simple Striker Video Links:

- Left Kick - Lab: https://youtu.be/9By_UIXWLzg
- Right Kick - Lab: https://youtu.be/H6pn1E-Xkdk
- Both Kicks - Simulation: https://youtu.be/YOyAr54ueso
- Left Dribble - Lab: https://youtu.be/d-HLVtI8HZI
- Right Dribble - Lab: https://youtu.be/SC-6XiMe9cY
- Both Dribbles - Simulation: https://youtu.be/f7QLn7aHafM

Simple Defender Video Links:

- Full Behavior - Lab: https://youtu.be/IaCwAt_2DIw
- Full Behavior - Simulation: https://youtu.be/rqOLmQZpWiU

Simple Goalkeeper Video Links:

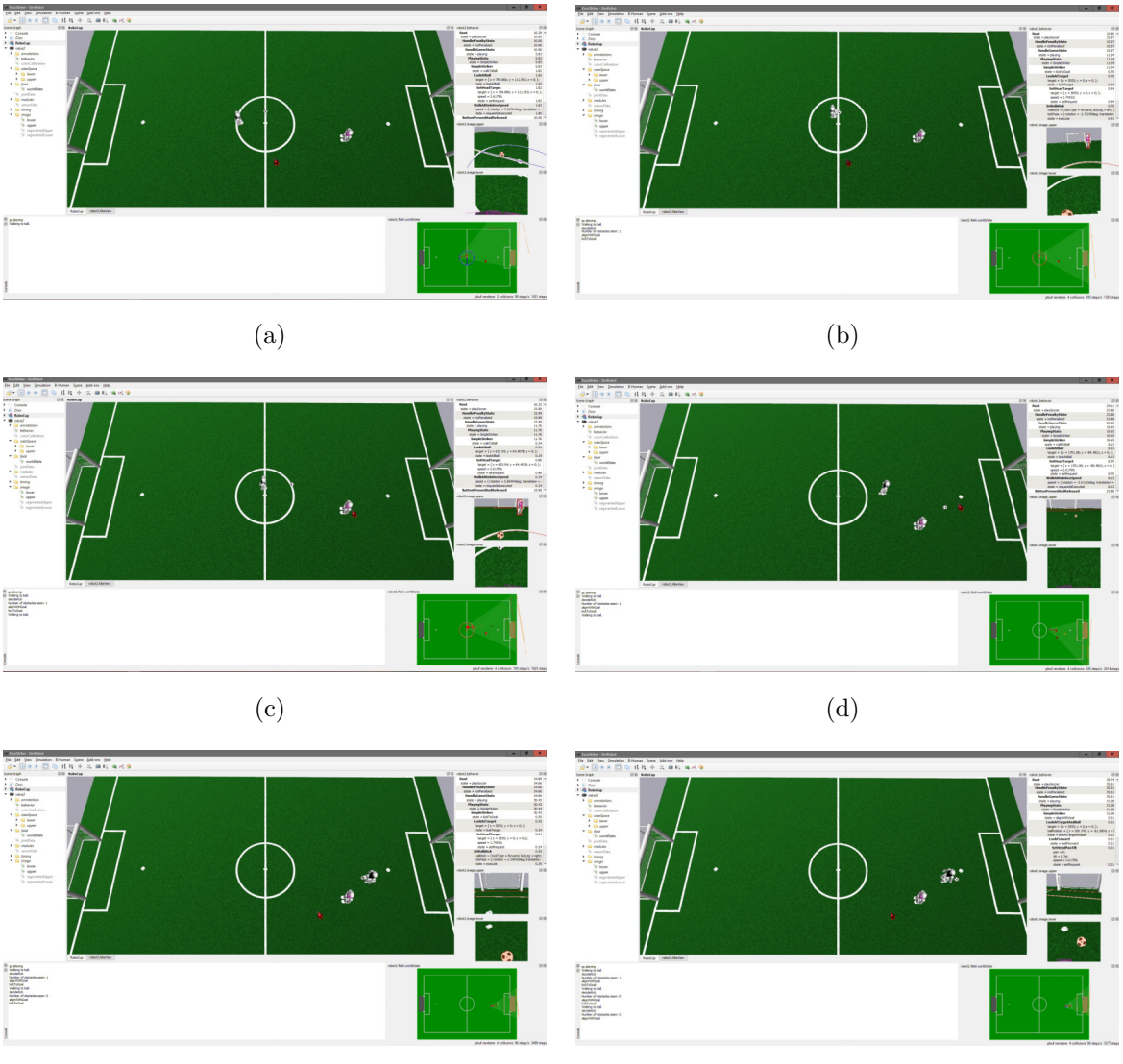
- Full Behavior - Lab: https://youtu.be/Mb9xCK873zs
- Full Behavior - Simulation: https://youtu.be/NHDf7qEkvdY
- Top Goal Approach - Simulation: https://youtu.be/4D5hM4Sb5nI
- Bottom Goal Approach - Simulation: https://youtu.be/mxUVfz6Nx3o
- Both Goal Approaches - Lab: https://youtu.be/hjFFTQ7TteU

Ball Control Video Links:

- Proper Alignment - Simulation: https://youtu.be/F3pdQlydZgA
- Both Ball Approaches - Simulation: https://youtu.be/R_1ZZrs-pcg
- Patrol - Simulation: https://youtu.be/EEuK2PzdXxs
- Proper Alignment - Lab: https://youtu.be/baaE_UzjQo4
- Patrol - Lab: https://youtu.be/jTPG64CwlZI

5.1 Simple Striker

Scenario no1: The striker heads towards the ball and kicks it straight to the opponent's goal since no obstacle intercepts its course as seen in Figure 5.1 and Figure 5.2. As mentioned in previous chapter, the kicking foot depends on the part of the field the ball is located.



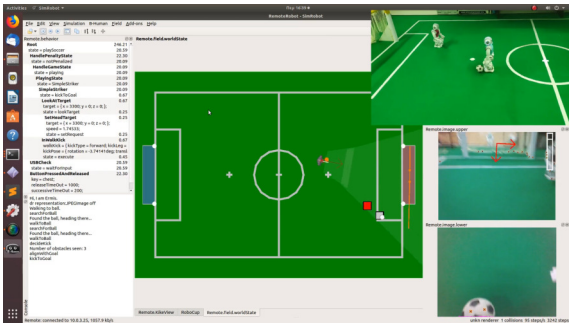
(a) (b)

(c) (d)

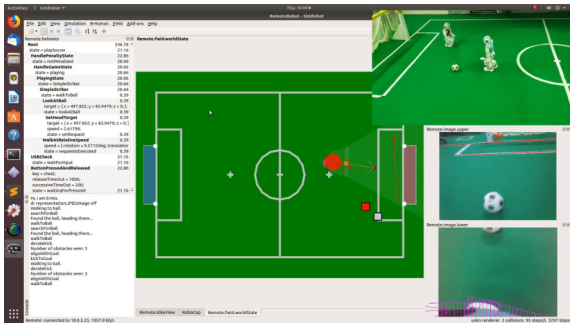
(e) (f)

Figure 5.1: Simple Striker: Simulation results of kicking the to ball towards the opponent goal. The appropriate kicking leg depends on the position of the ball.

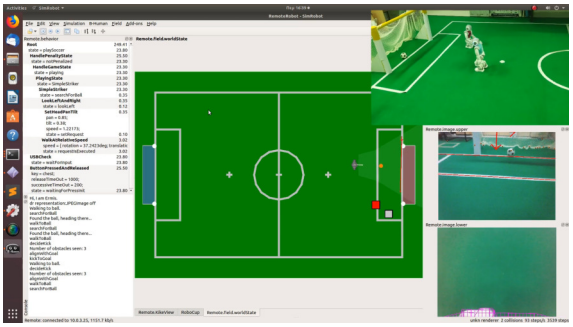
(a)



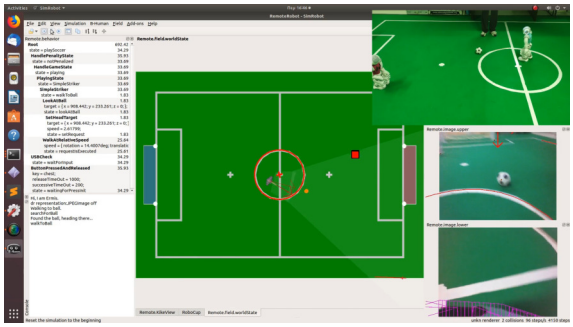
(b)



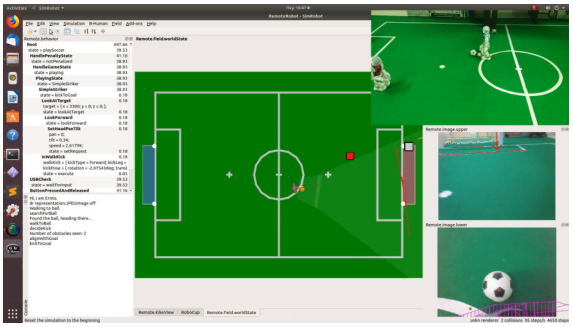
(c)



(d)

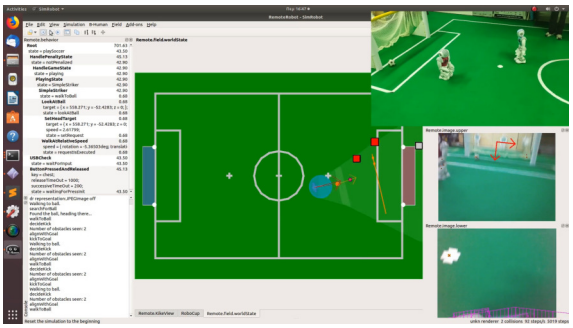


(e)



(f)

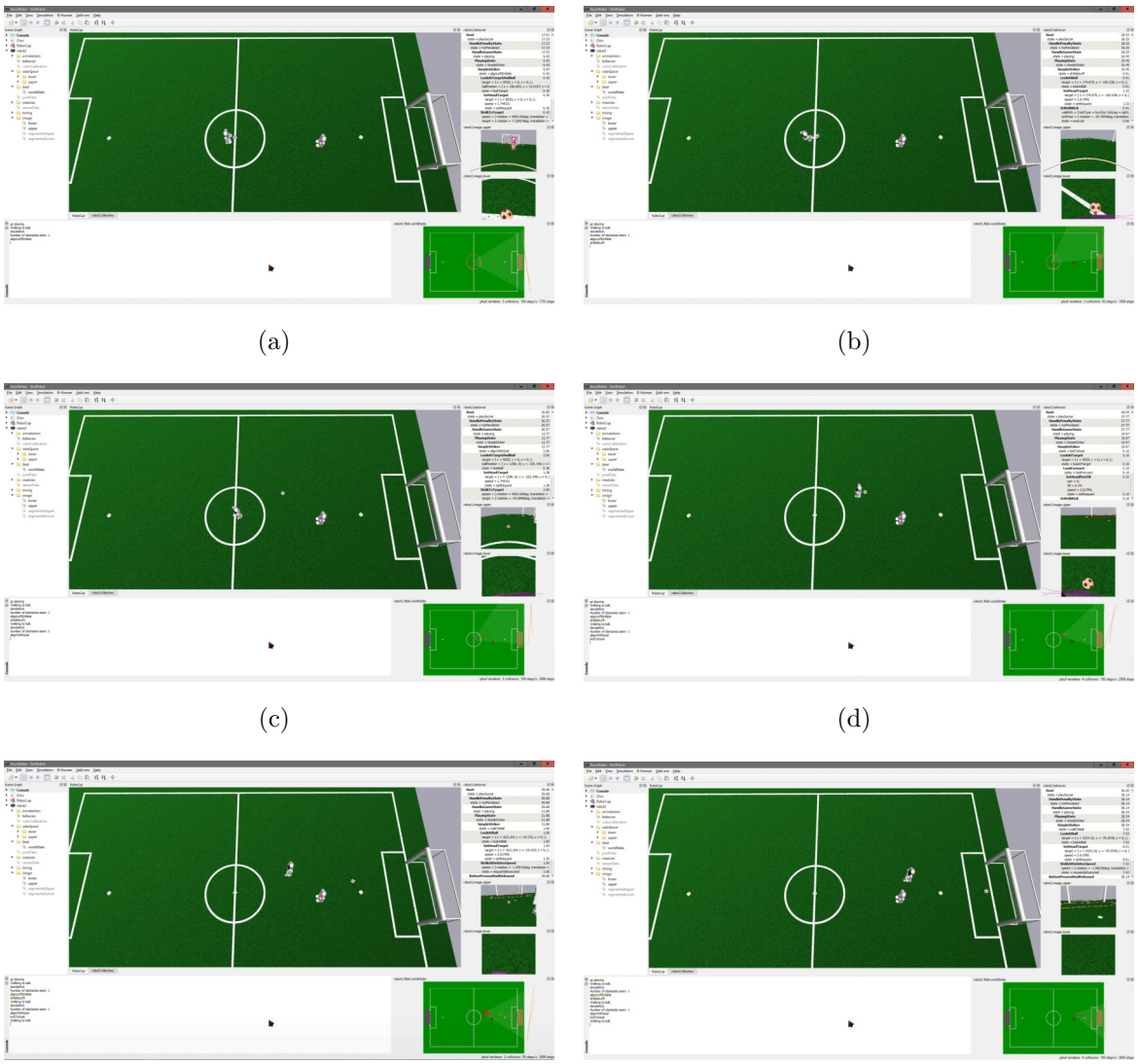
(g)



(h)

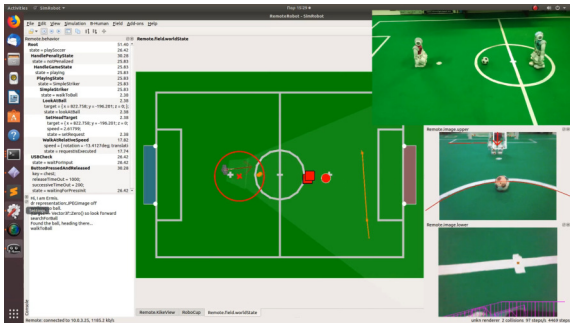
Figure 5.2: Simple Striker: Lab results of kicking the to ball towards the opponent goal. The appropriate kicking leg depends on the position of the ball.

Scenario no2: The striker heads towards the ball and executes a left dribble as seen in Figure 5.3 and Figure 5.4 in order to avoid the opponent that blocks its course to the goal. The left dribble is achieved with the use of the right leg so the striker needs to align accordingly in order to use that leg effectively.

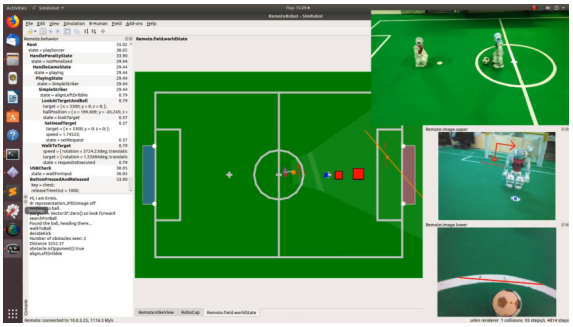


(a) (b) (c) (d) (e) (f)

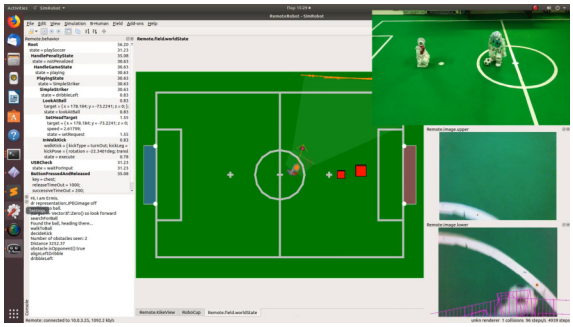
Figure 5.3: Simple Striker: Simulation results of left dribble.



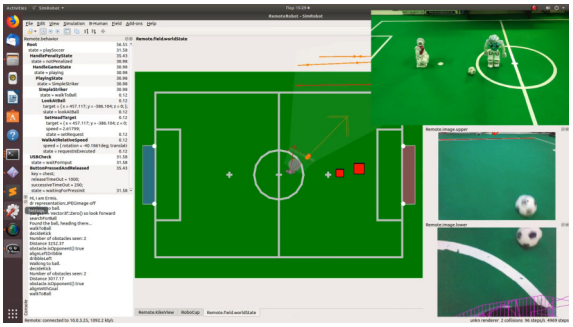
(a)



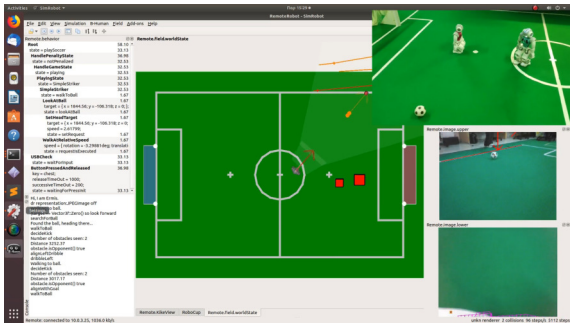
(b)



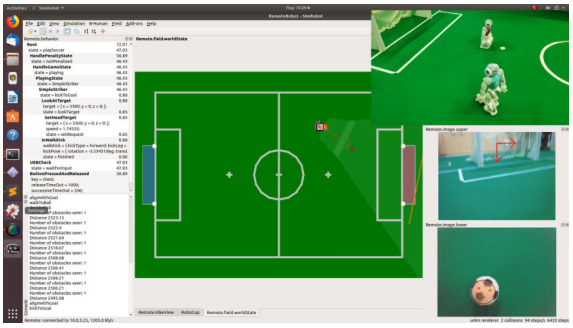
(c)



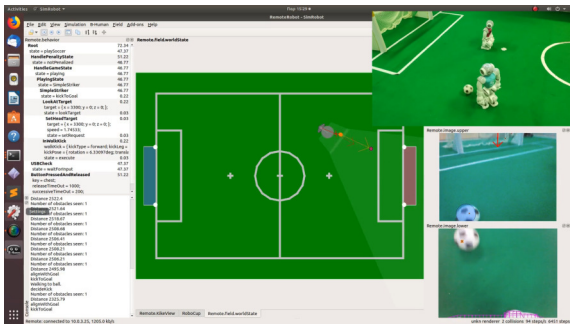
(d)



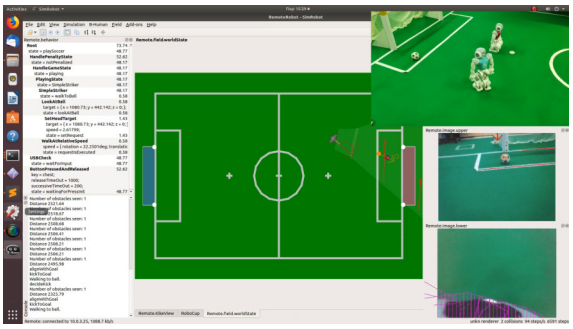
(e)



(f)



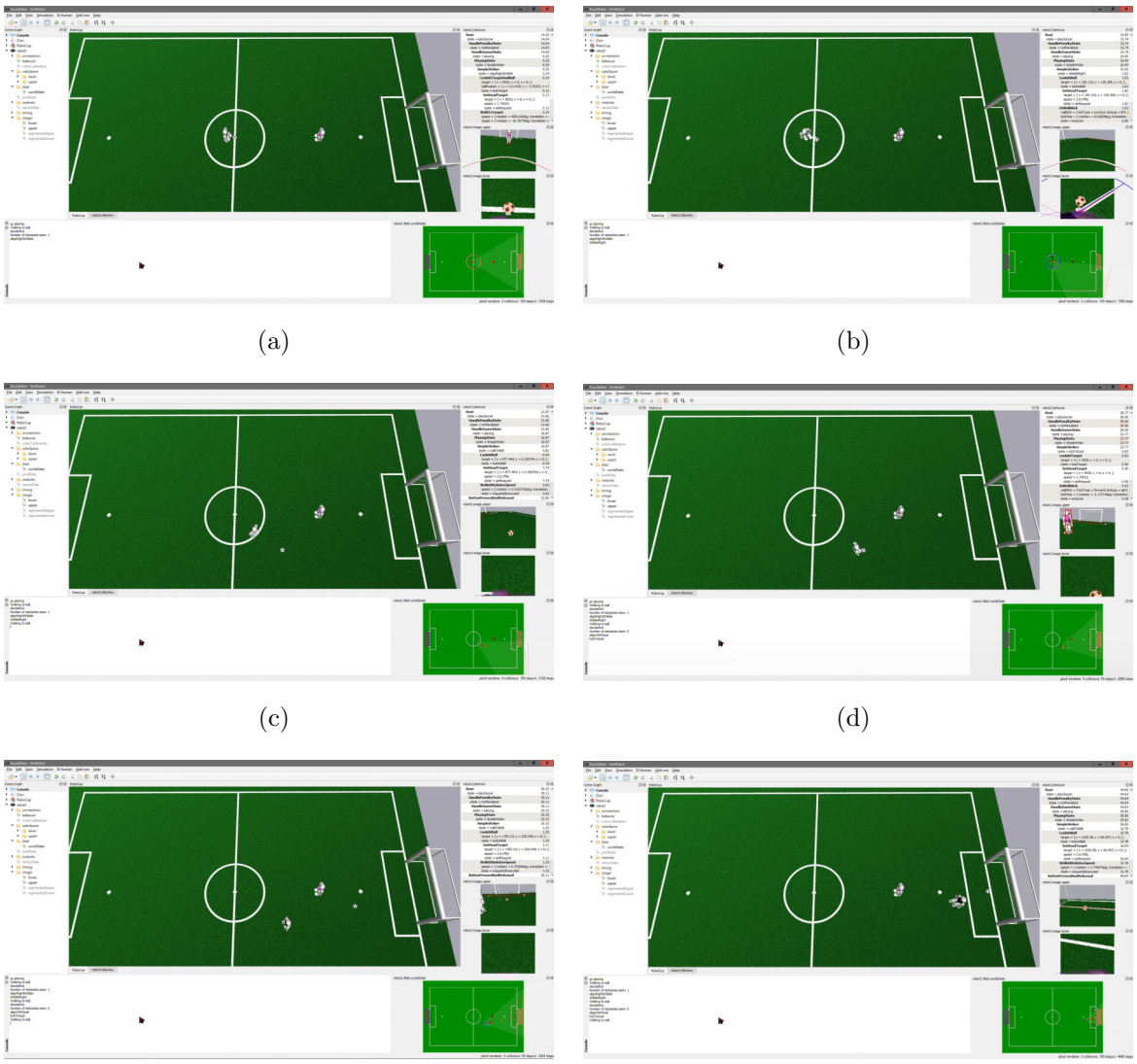
(g)



(h)

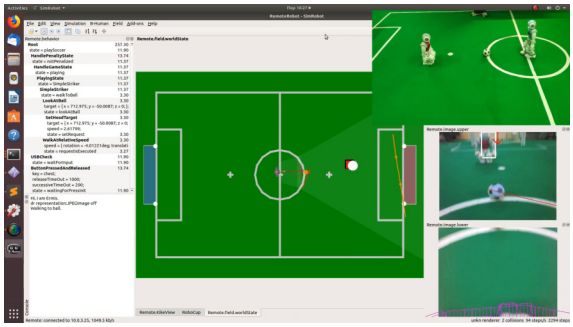
Figure 5.4: Simple Striker: Lab results of left dribble.

Scenario no3: The striker heads towards the ball and executes a right dribble as seen in Figure 5.5 and Figure 5.6, based on where the obstacle between it and the goal is located. This dribble utilizes the left foot along with the proper alignment executed before the dribble. While the two dribbles are mirrored it seemed that in both environments, the left dribble was less powerful.

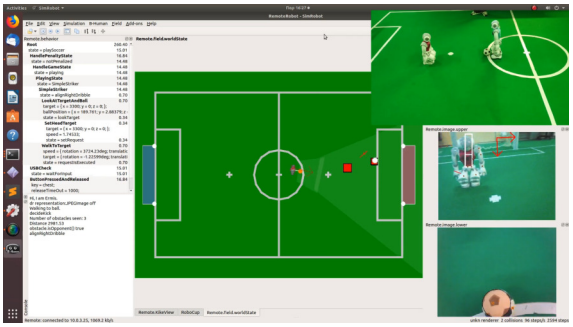


(a) (b) (c) (d) (e) (f)

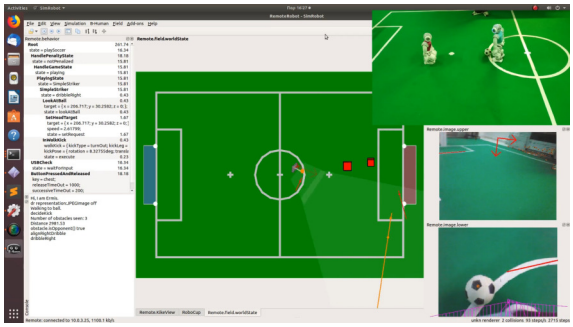
Figure 5.5: Simple Striker: Simulation results of right dribble.



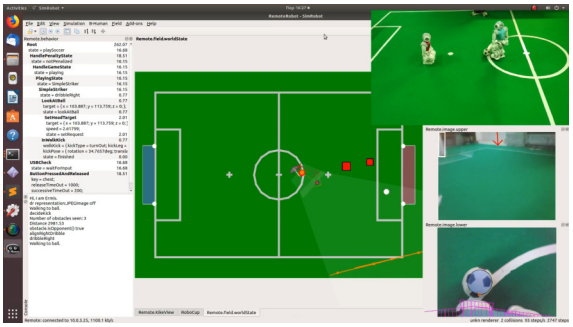
(a)



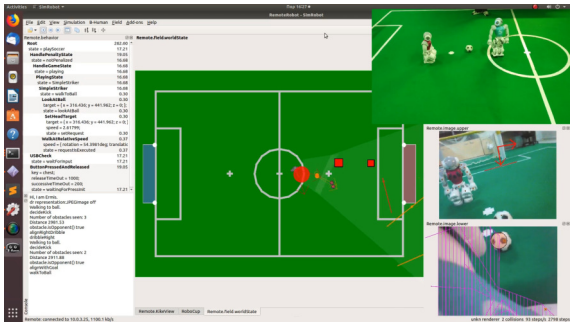
(b)



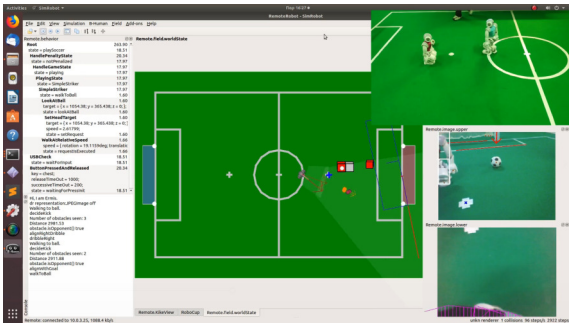
(c)



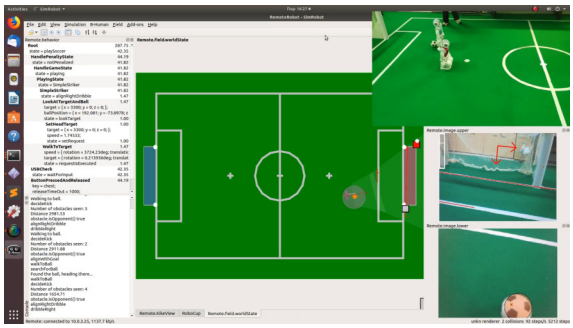
(d)



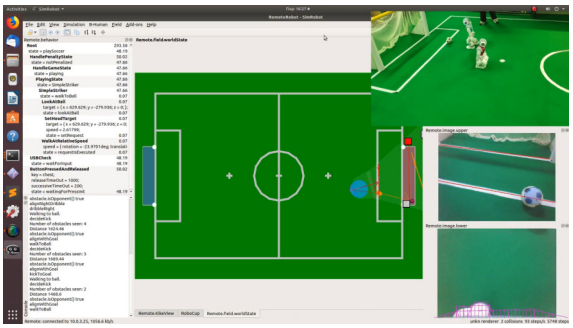
(e)



(f)



(g)

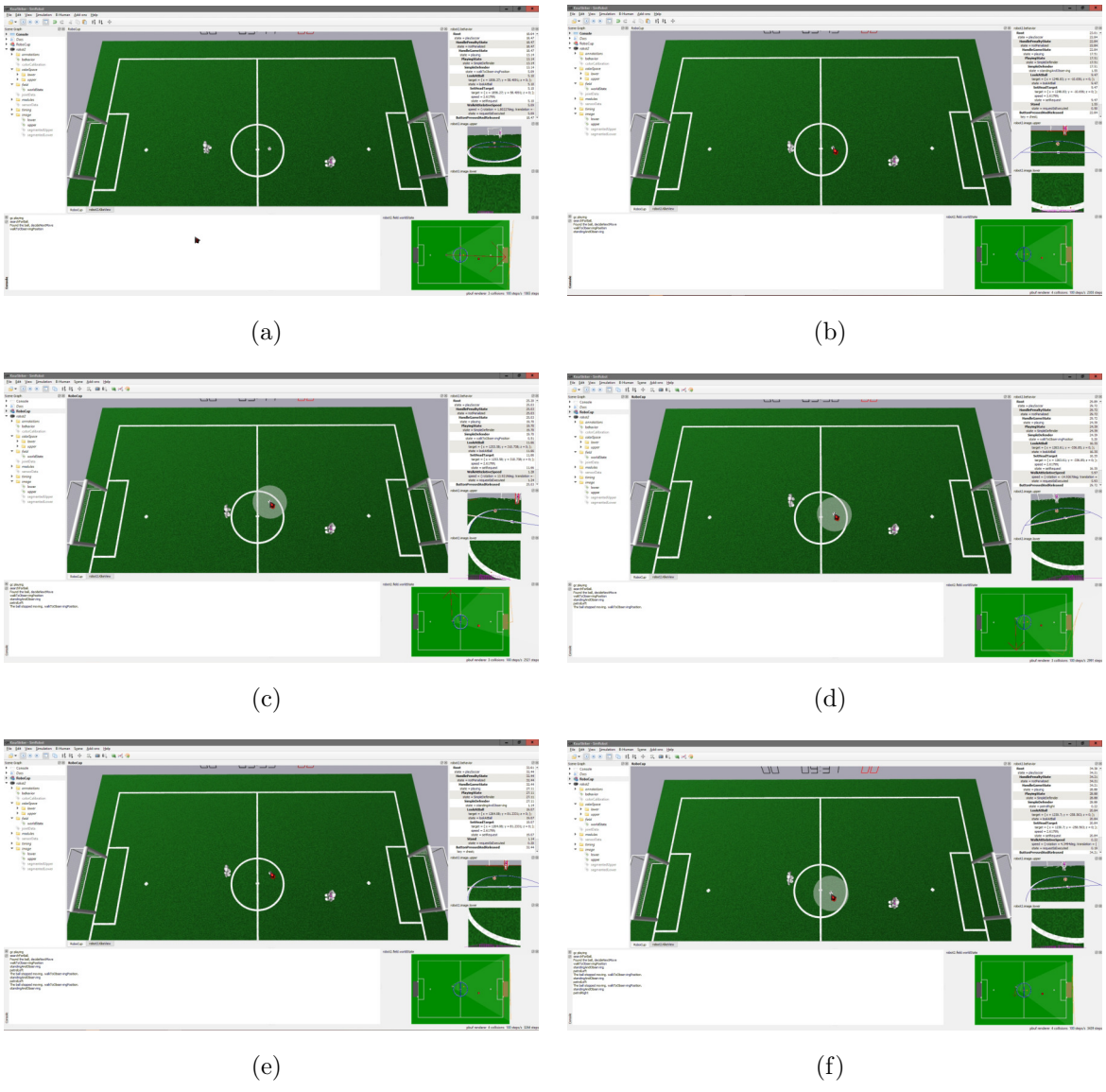


(h)

Figure 5.6: Simple Striker: Lab results of right dribble.

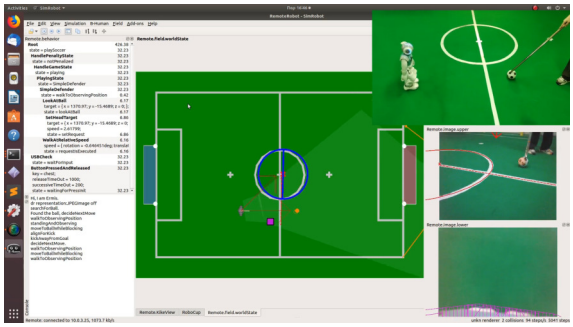
5.2 Simple Defender

Scenario no1: Staying in his side of the field, the defender is trying to match his y coordinate with the global y coordinate of the ball. That way, it blocks any ball movement towards its own half of the field and in the same time stays close to the ball in order to prevent any threatening move towards its goal. The results of this behavior can be seen in Figure 5.7 and Figure 5.8.

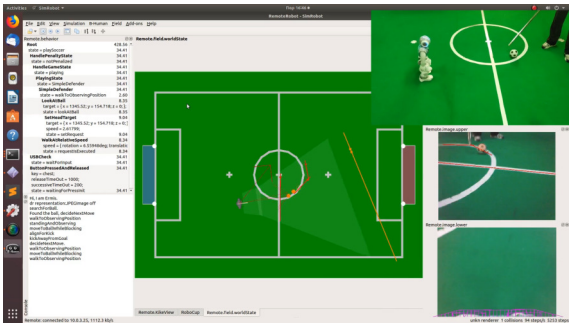


(a) (b) (c) (d) (e) (f)

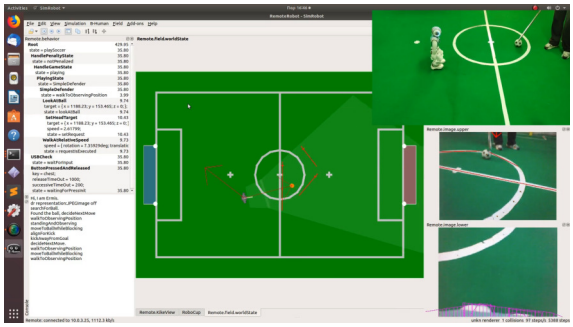
Figure 5.7: Simple Defender: Simulation results of patrol movement.



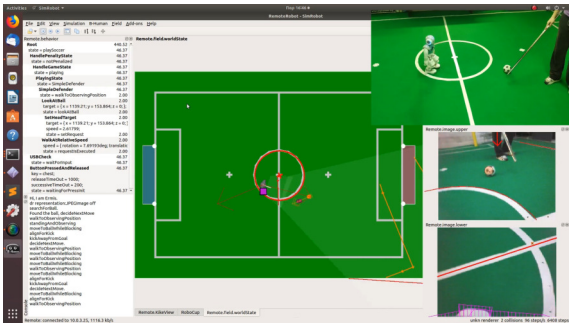
(a)



(b)



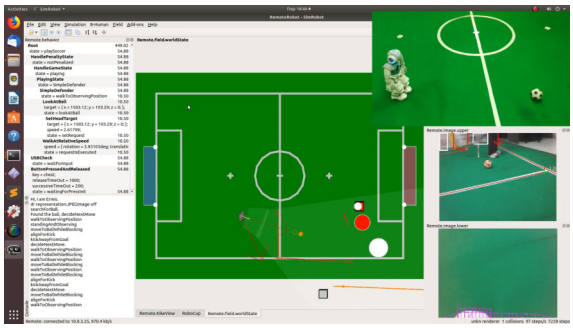
(c)



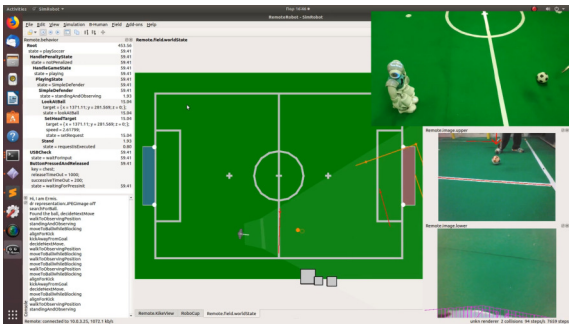
(d)

(e)

(f)



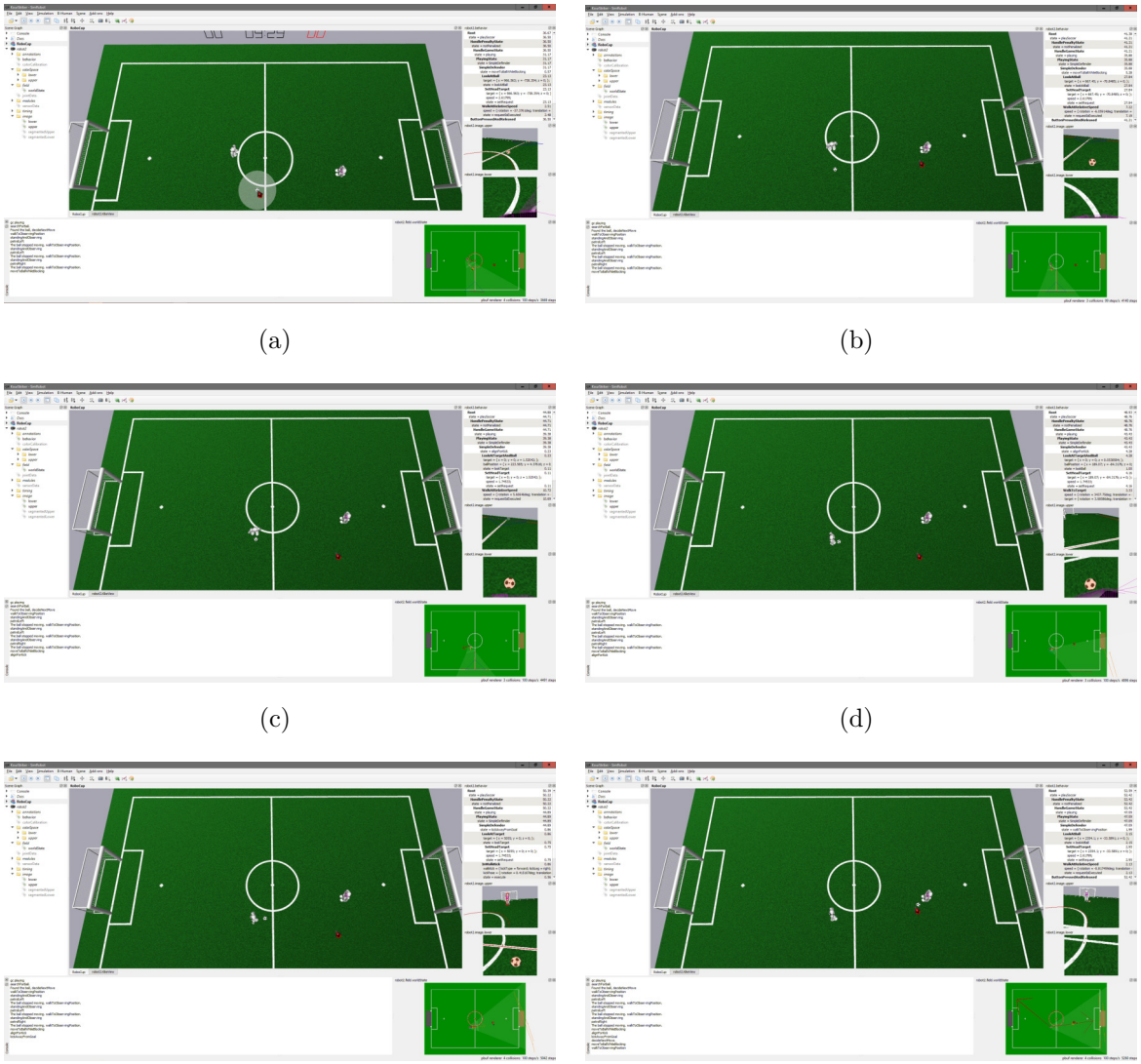
(g)



(h)

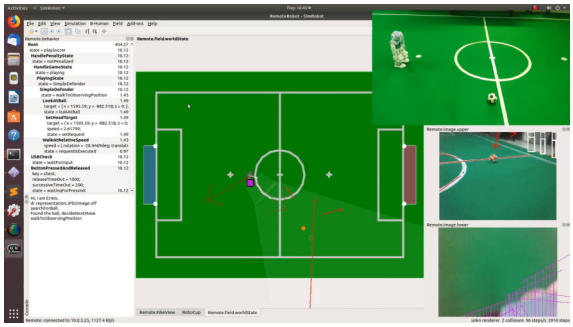
Figure 5.8: Simple Defender: Lab results of patrol movement.

Scenario no2: The ball is located at the defender's team side of the field. At this location it is considered a potential threat for its goal hence the defender proceeds into walking towards it and kicking it away towards the opponent's goal as seen in Figure 5.9 and Figure 5.10. As soon as the ball moves away from its side of the field it moves back to observing the ball and aligning with it properly.

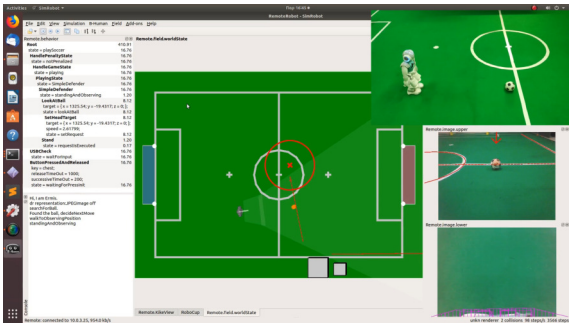


(a) (b) (c) (d) (e) (f)

Figure 5.9: Simple Defender: Simulation results of kick away motion.



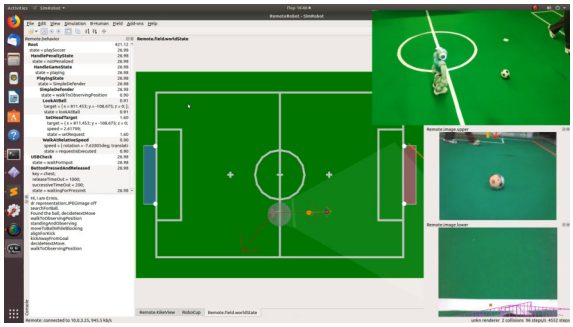
(a)



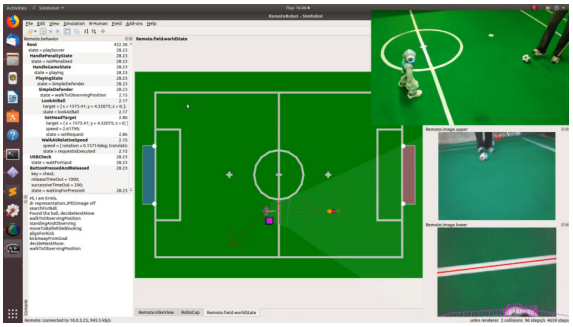
(b)

(c)

(d)

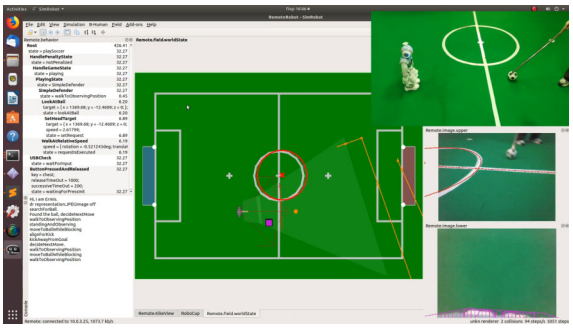


(e)



(f)

(g)



(h)

Figure 5.10: Simple Defender: Lab results of kick away motion.

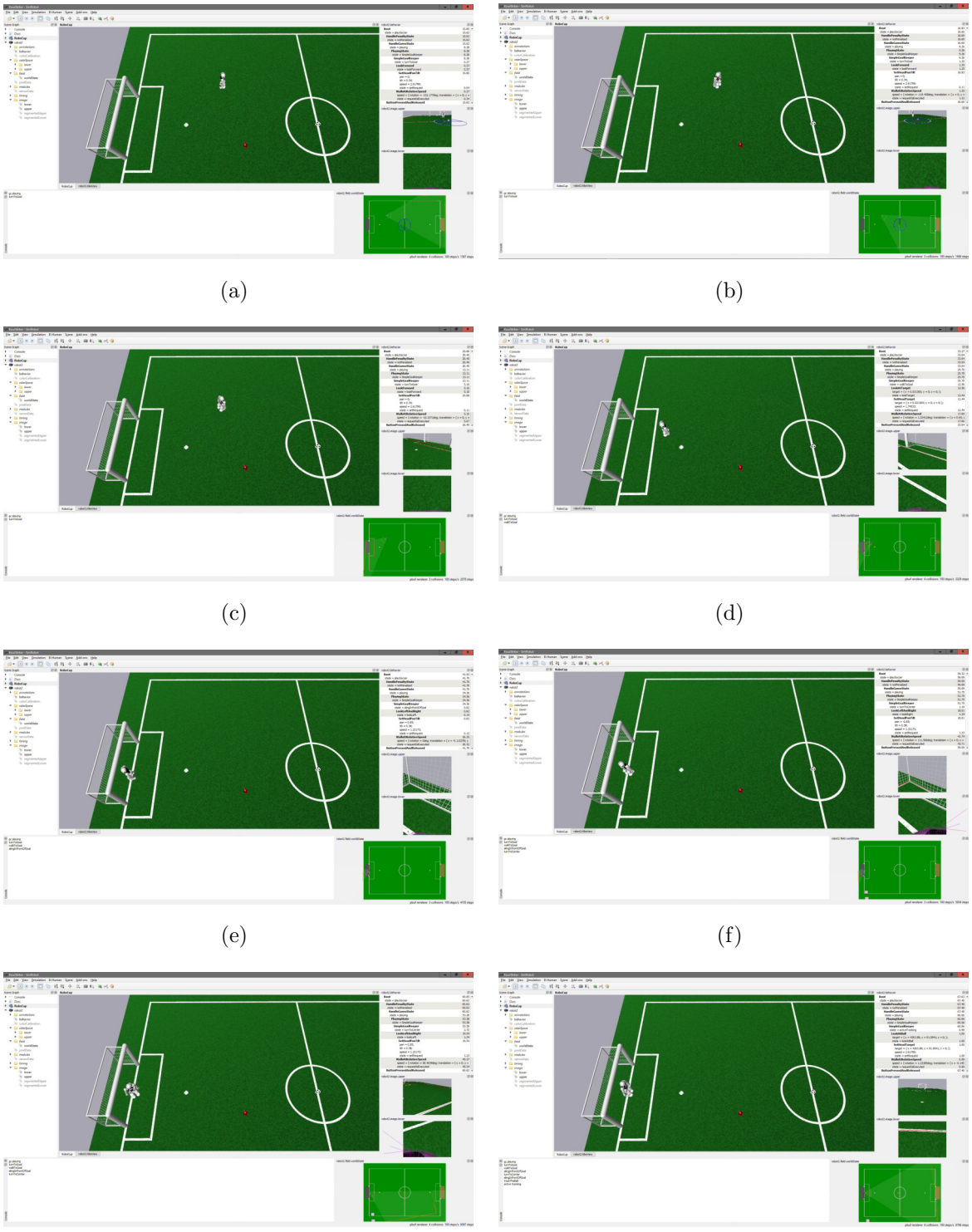
5.3 Simple Goalkeeper

Scenario no1: The goal keeper heads towards its goal from the top part of the field and position itself properly as seen in Figure 5.11 and Figure 5.12. This part of the behavior that is responsible for the proper positioning of the goalkeeper in addition to any other role is really important cause it occurs multiple times during a game. It occurs at the start of the game and during the game if a robot is penalized.

Scenario no2: The goal keeper heads to its goal from the bottom part of the field and position itself properly as seen in Figure 5.13 and Figure 5.14.It is important to note that in both cases, top and bottom approach, the goalkeeper is able to determine its position and chose the shortest path that will align it first with the goal and after with the center of the field.

Scenario no3: The goalkeeper is in position in front of its goal and the ball is on its own side of the field. When the ball moves sideways the goalkeeper moves as well in order to protect its goal and also kicks it away when it enters the penalty area as seen in Figure 5.15 and Figure 5.16.

Scenario no4: The goalkeeper is in position in front of its goal and the ball is on its own side of the field. When the ball moves sideways the goalkeeper moves as well in order to protect the open corner but stops at a specific point in order to protect its goal more efficiently and also to avoid any colision with the goal posts as seen in Figure 5.17 and Figure 5.18.

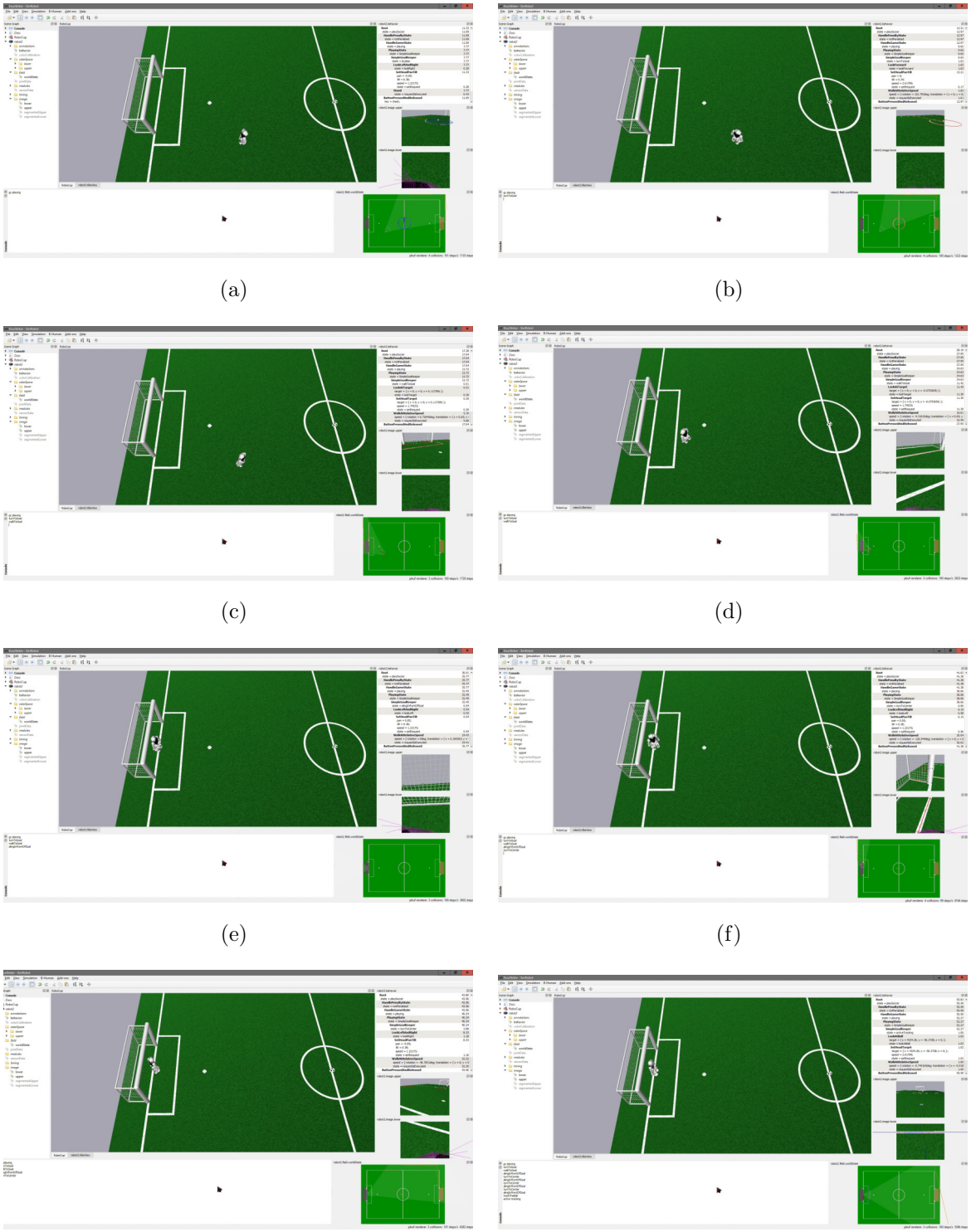


(a) (b) (c) (d) (e) (f) (g) (h)

Figure 5.11: Simple Goalkeeper: Simulation results of heading to goal and aligning motion from the top part of the field.

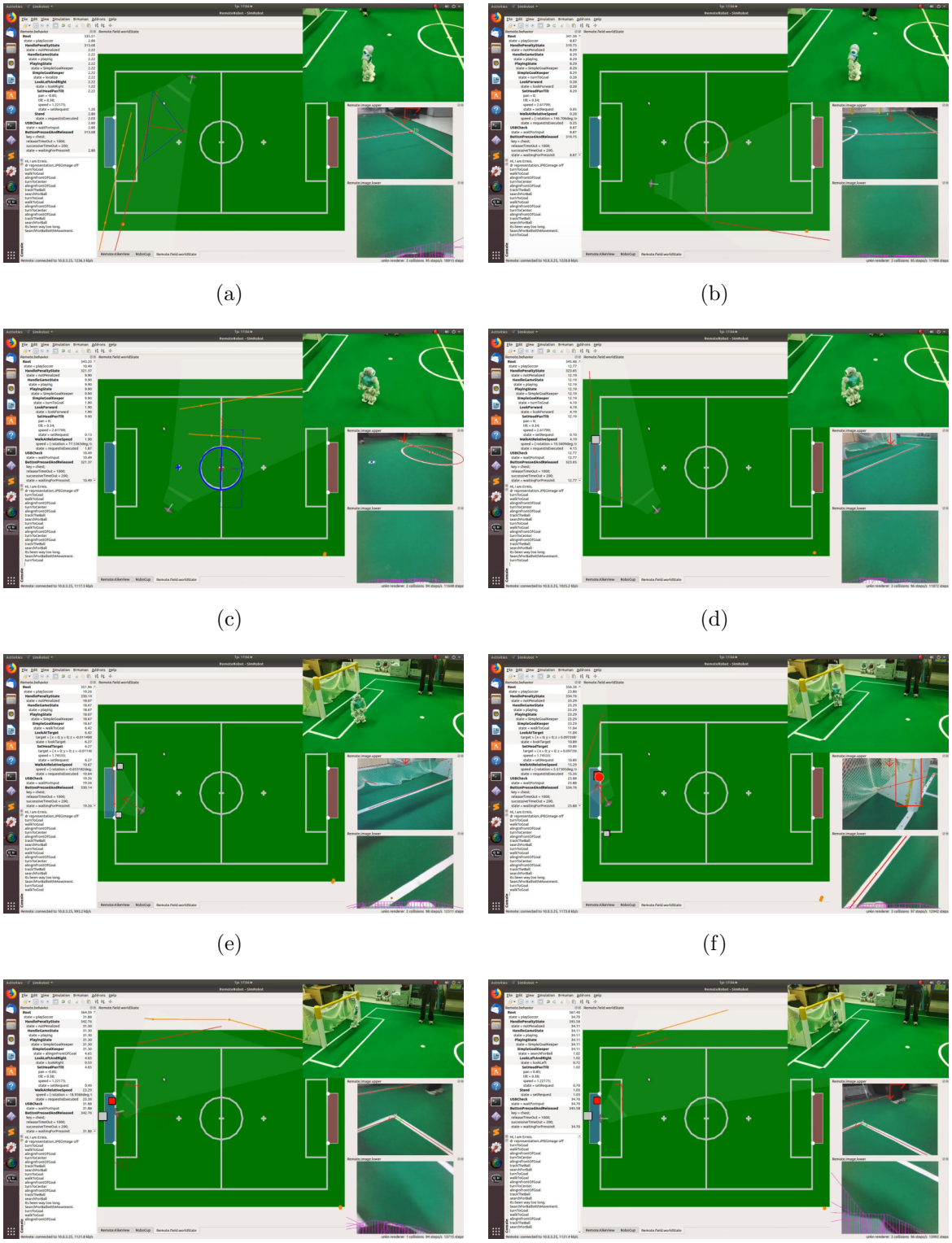
(a) (b) (c) (d) (e) (f) (g) (h)

Figure 5.12: Simple Goalkeeper: Lab results of heading to goal and aligning motion from the top part of the field.



(a) (b) (c) (d) (e) (f) (g) (h)

Figure 5.13: Simple Goalkeeper: Simulation results of heading to goal and aligning motion from the lower part of the field.



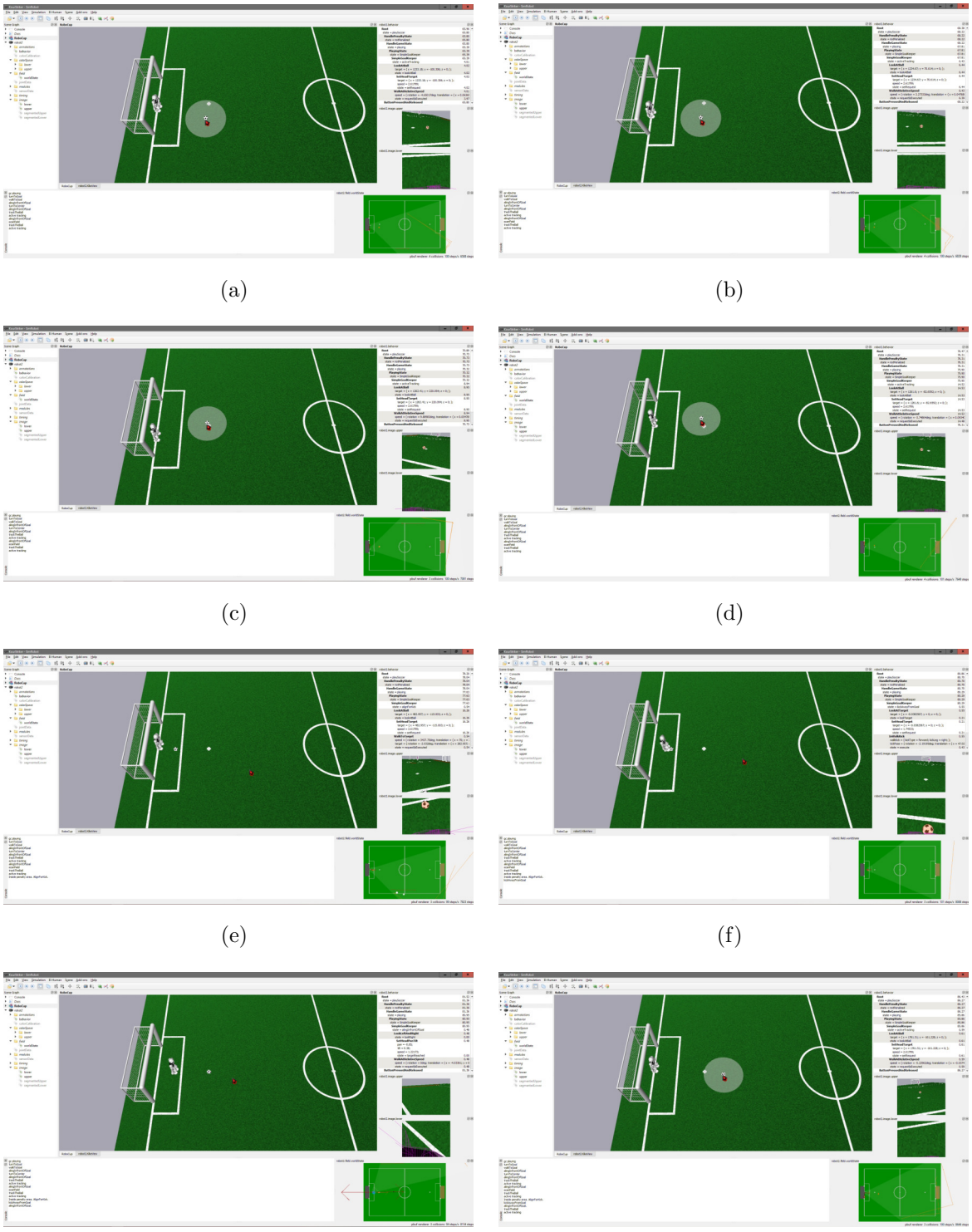
(a) (b)

(c) (d)

(e) (f)

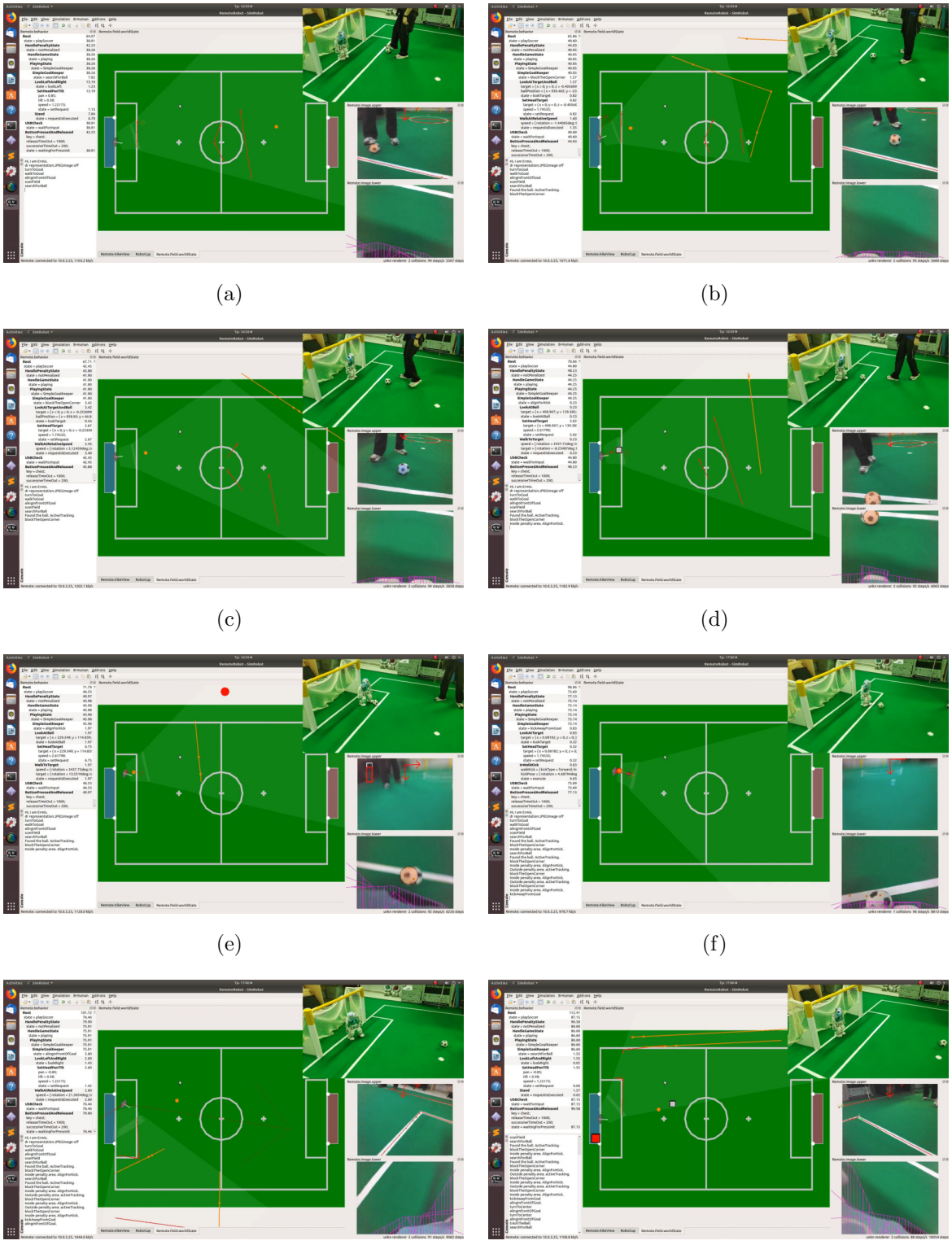
(g) (h)

Figure 5.14: Simple Goalkeeper: Lab results of heading to goal and aligning motion from the lower part of the field.



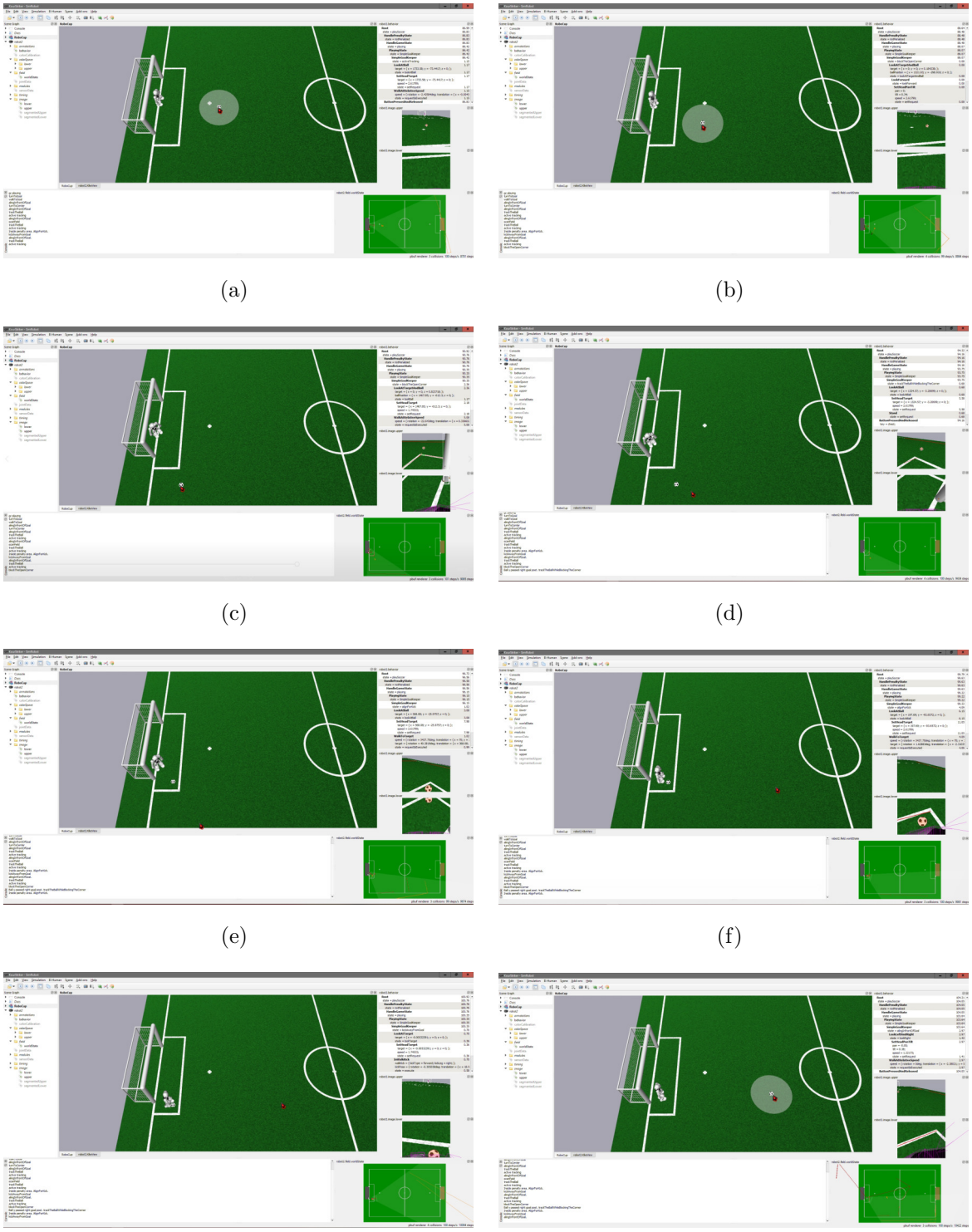
(a) (b) (c) (d) (e) (f) (g) (h)

Figure 5.15: Simple Goalkeeper: Simulation results of the goalkeeper active tracking and kicking away.



(a) (b) (c) (d) (e) (f) (g) (h)

Figure 5.16: Simple Goalkeeper: Lab results of the goalkeeper active tracking and kicking away.



(a) (b) (c) (d) (e) (f) (g) (h)

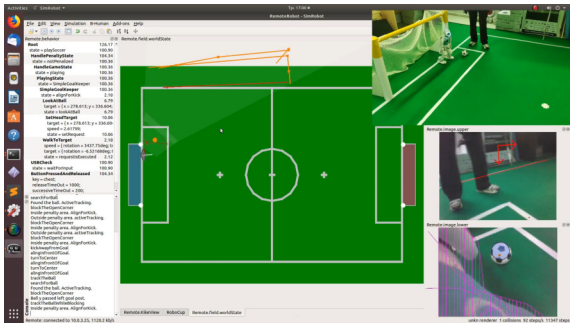
Figure 5.17: Simple Goalkeeper: Simulation results of the goalkeeper movements blocking the corners.

(a)

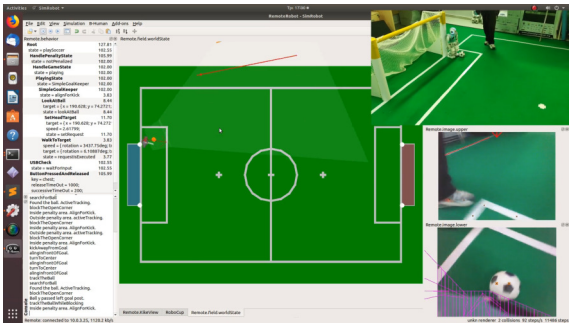
(b)

(c)

(d)



(e)



(f)

(g)

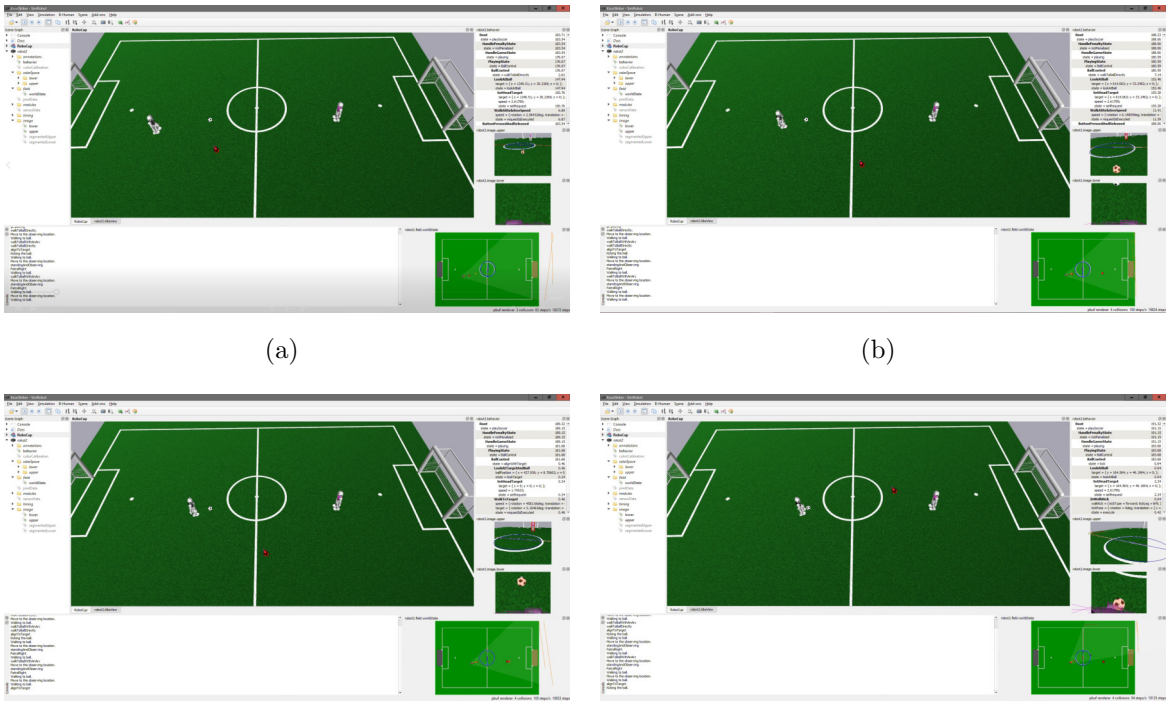
(h)

Figure 5.18: Simple Goalkeeper: Lab results of the goalkeeper movements blocking the corners.

5.4 BallControl

Scenario no1: In this scenario the ball is not located inside the center circle. In addition, it is located in such a way that it can be handled with the `walkToBallDirectly`. After reaching the ball, the robot aligns and kicks it towards the target as seen in Figure 5.19 and Figure 5.20.

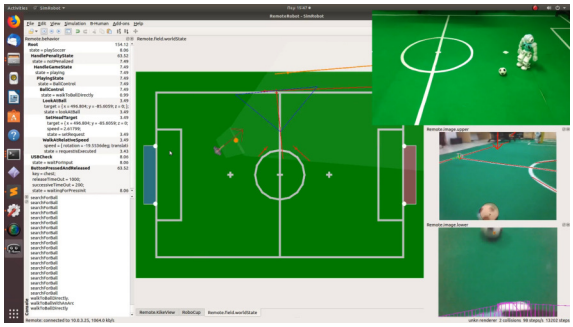
Scenario no2: The ball is not located inside the center circle but is located in such a way that the robot would need to execute a long alignment movement if it walked to the ball directly. Instead, it proceeds to handle the approach with the `walkToBallWithAnArc`. After reaching the ball, executes the final alignment and kicks it towards the target as seen in Figure 5.21 and Figure 5.22.



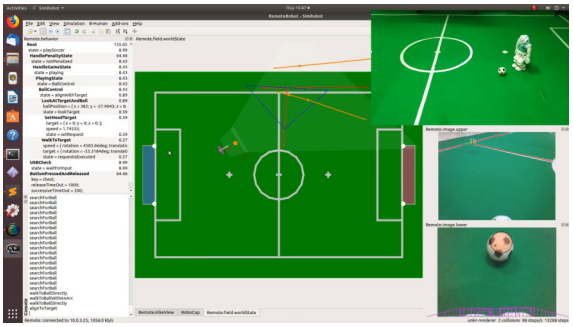
(a) (b)

(c) (d)

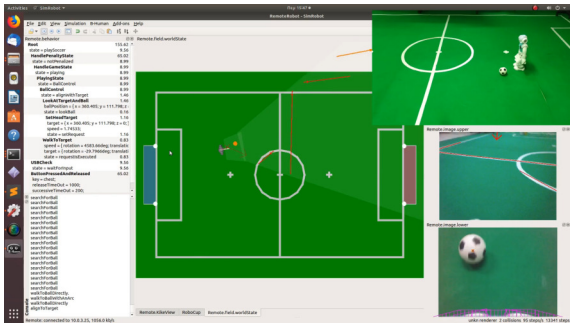
Figure 5.19: Ball Control: Simulation results of walking to ball directly.



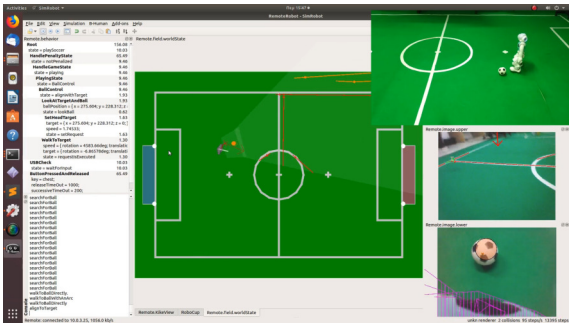
(a)



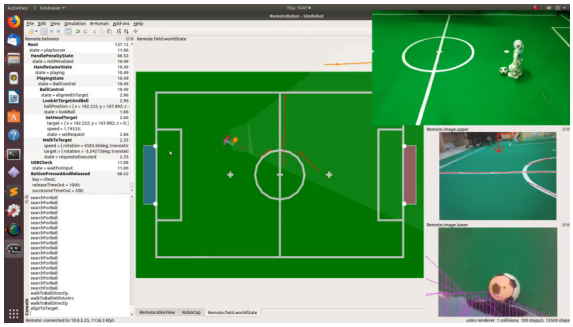
(b)



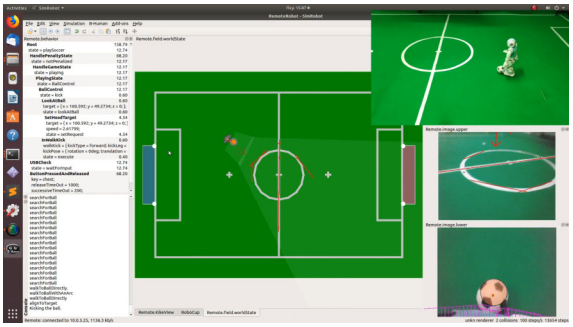
(c)



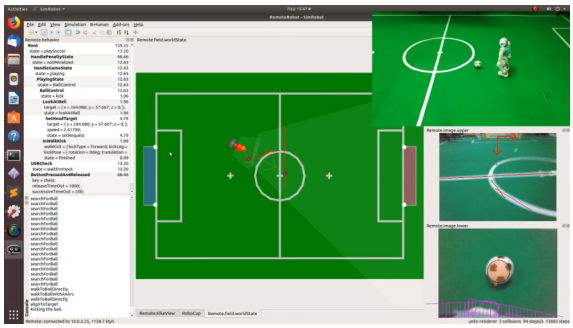
(d)



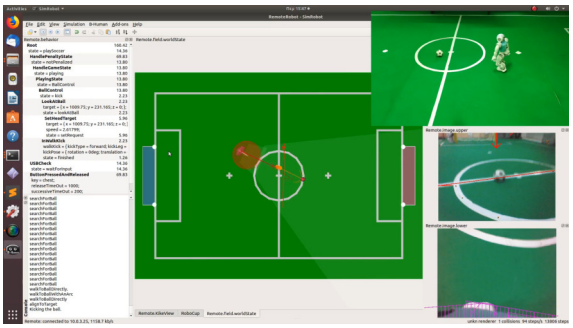
(e)



(f)

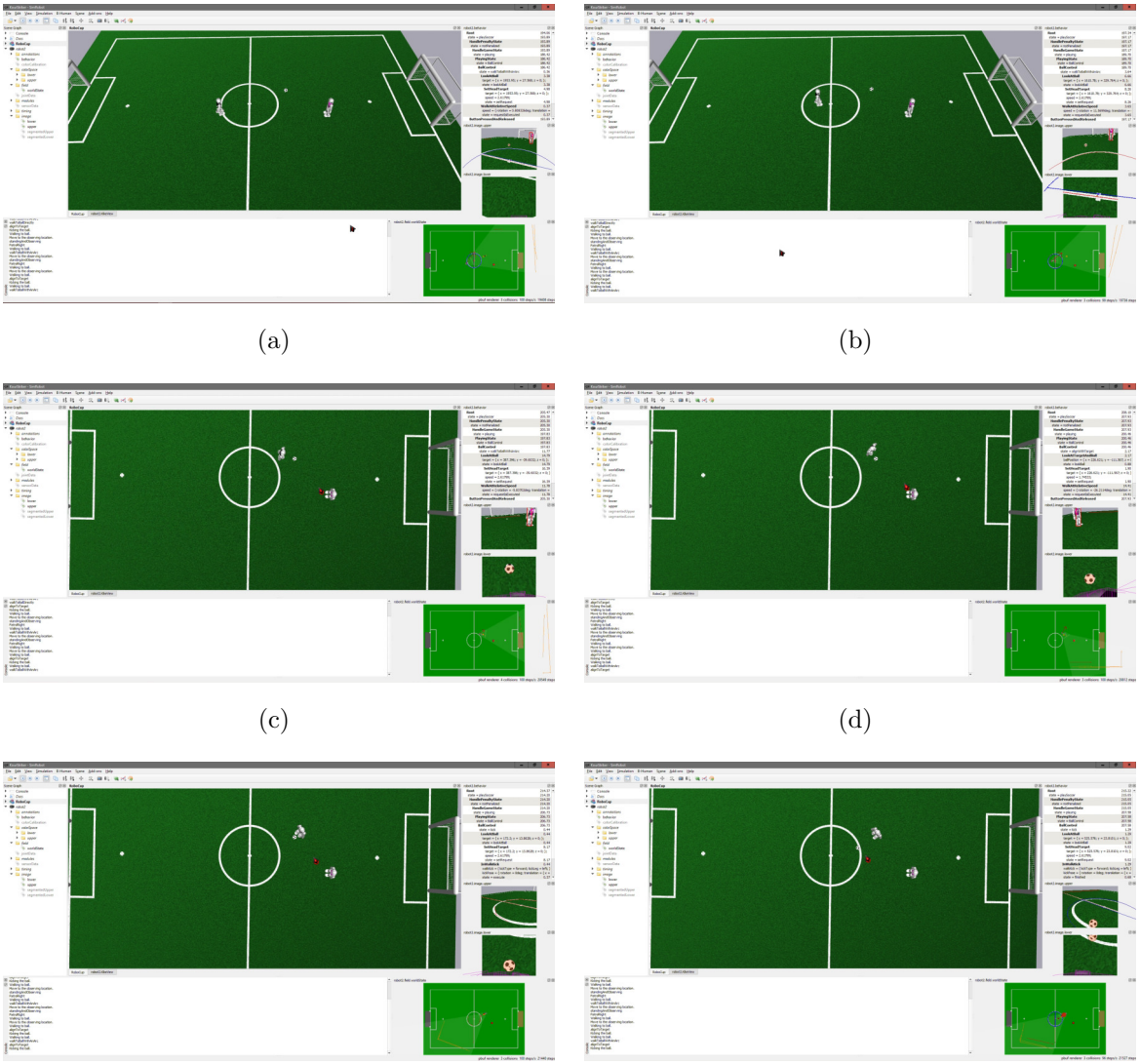


(g)



(h)

Figure 5.20: Ball Control: Lab results of walking to ball directly.

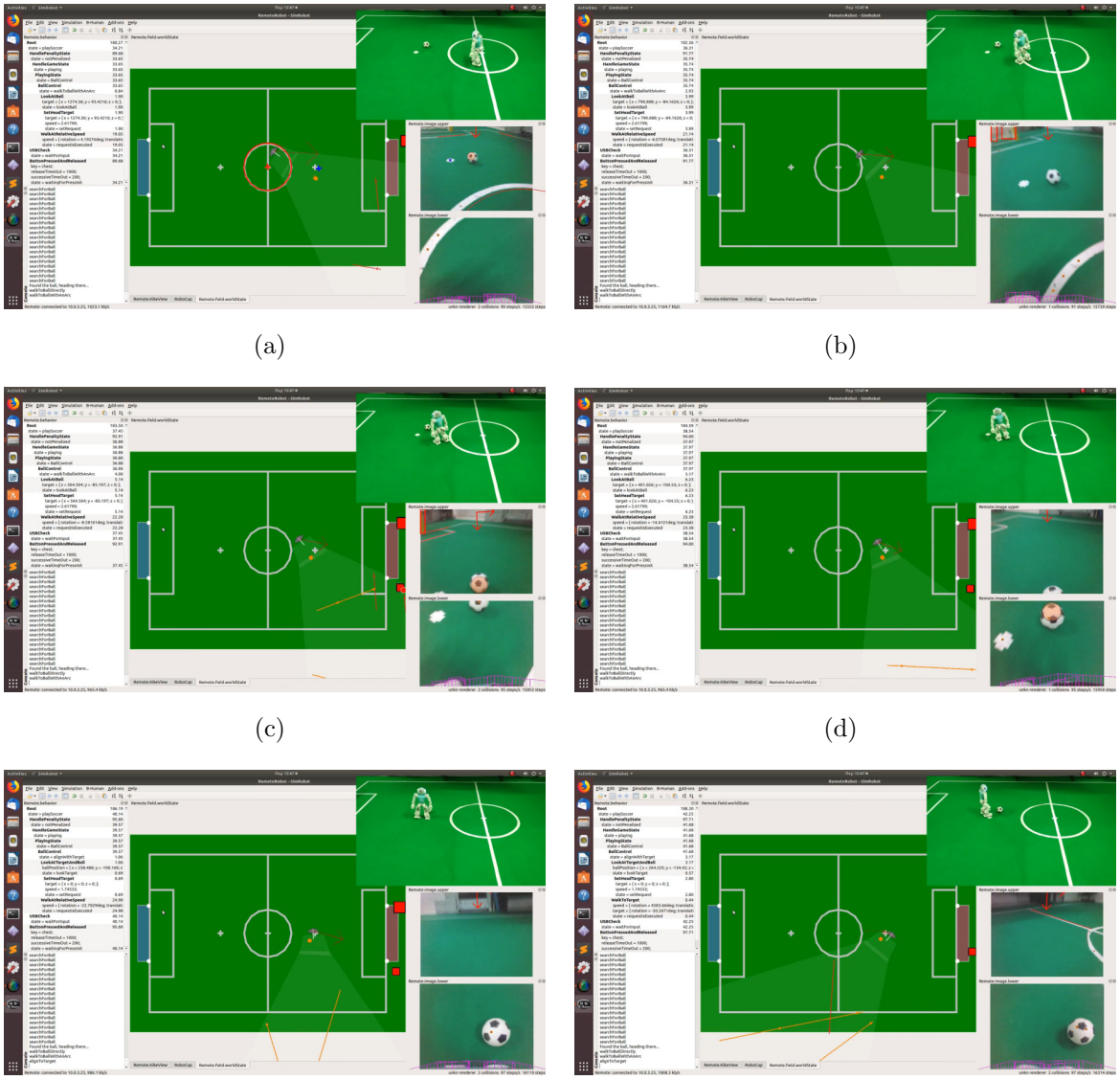


(a) (b)

(c) (d)

(e) (f)

Figure 5.21: Ball Control: Simulation results of walking to ball with an arc.



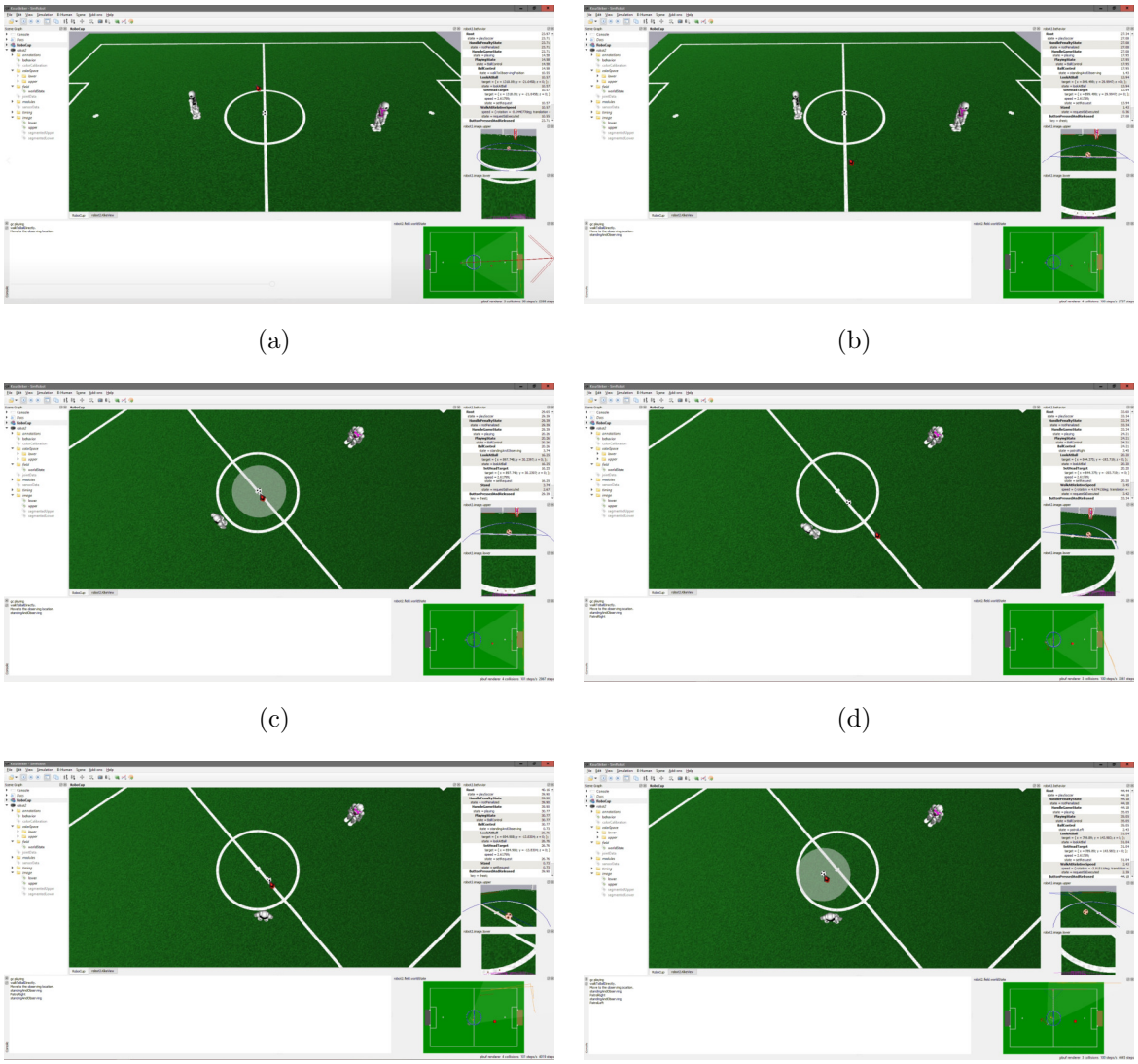
(a) (b)

(c) (d)

(e) (f)

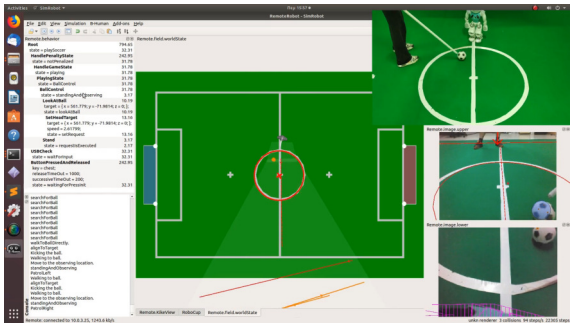
Figure 5.22: Ball Control: Lab results of walking to ball with an arc.

Scenario no3: The ball is located inside the center circle as soon as the robot finishes the localization. After the robot has reached the observing position, the ball starts moving inside the circle triggering the patrol movement of the robot as seen in Figure 5.23 and Figure 5.24. The patrol movement depends on the velocity of the ball.

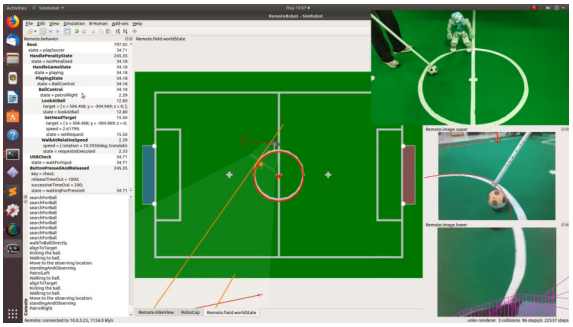


(a) (b) (c) (d) (e) (f)

Figure 5.23: Ball Control: Simulation results of patrolling sideways.



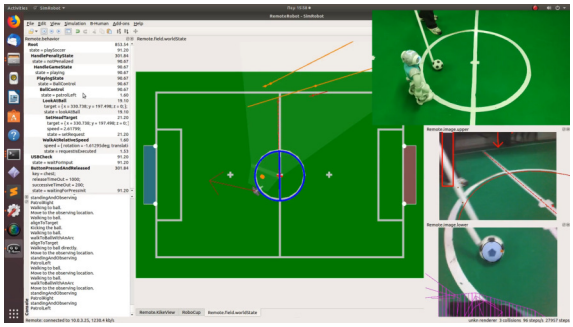
(a)



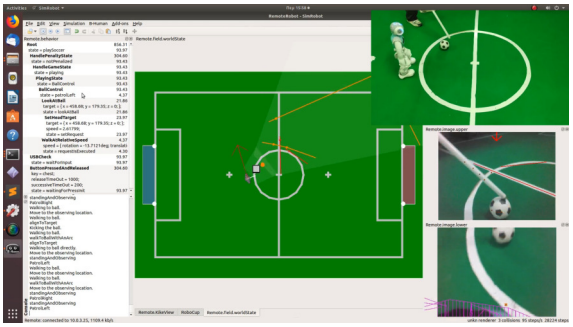
(b)

(c)

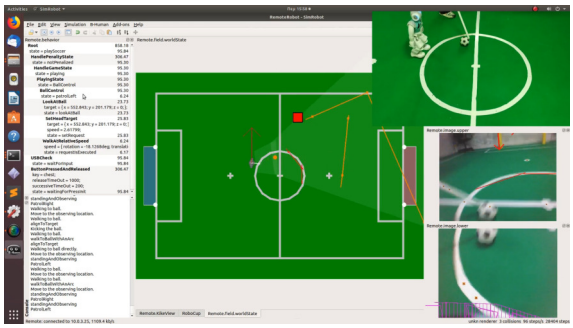
(d)



(e)



(f)

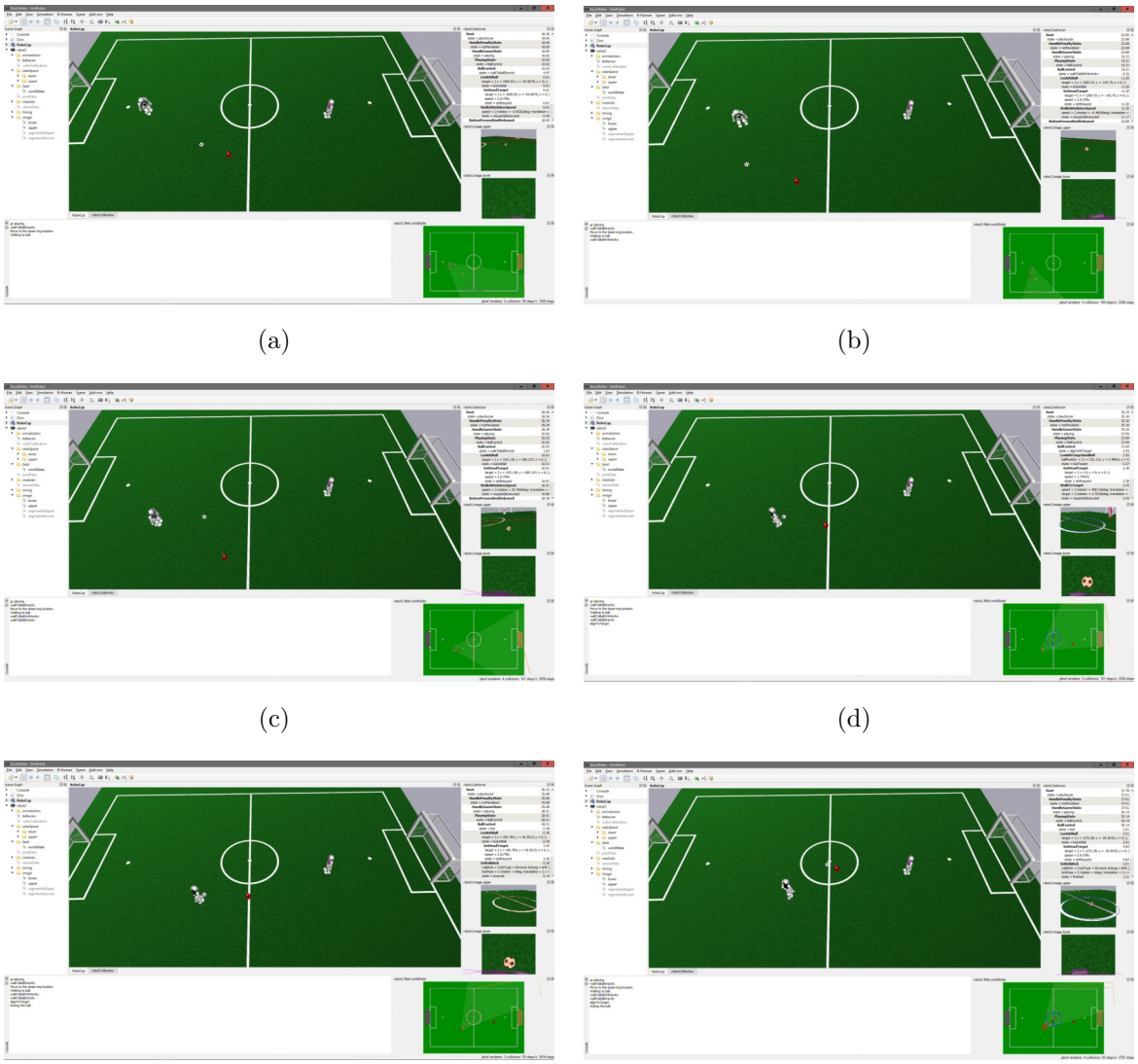


(g)

(h)

Figure 5.24: Ball Control: Lab results of patrolling sideways.

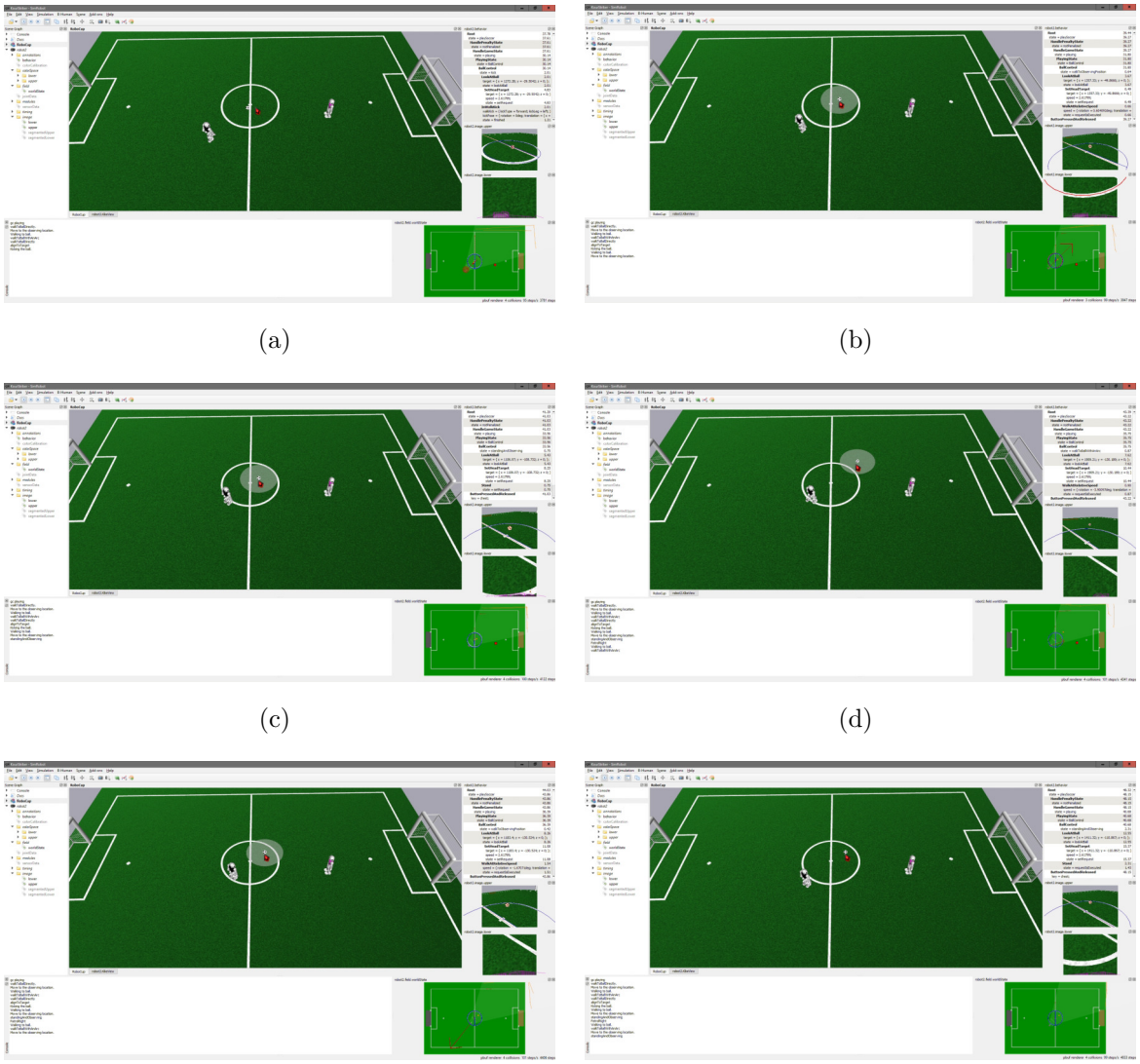
Scenario no4: The ball is not located inside the center circle but is located in a way that it must be handled with the `walkToBallWithAnArc`. While the robot walks towards the ball, the ball is being moved to a different position that does not require angled approach anymore. This scenario can be seen in Figure 5.25. The same occurs in the lab but we are unable to represent it with images due to the angle of the shooting.



(a) (b) (c) (d) (e) (f)

Figure 5.25: Ball Control: Simulation results of the proper alignment with the ball.

Scenario no5: The ball is located inside the target, in our case the center circle. As the robot tracks it , it is removed briefly and then positioned back in the circle. Our robot starts walking towards the ball when it moves but as soon as we position it back on target, it returns to its position as seen in Figure 5.26 and Figure 5.27.

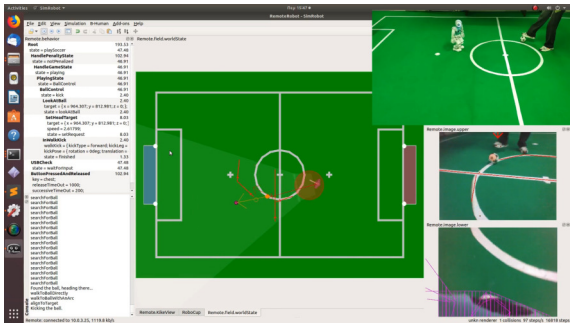


(a) (b)

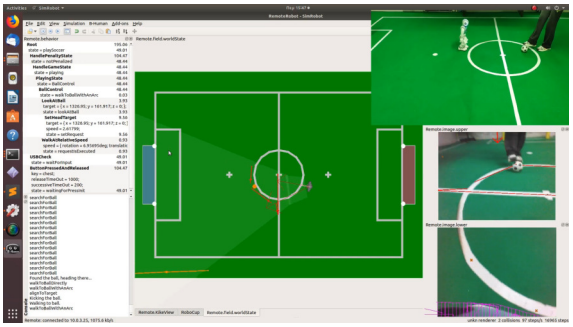
(c) (d)

(e) (f)

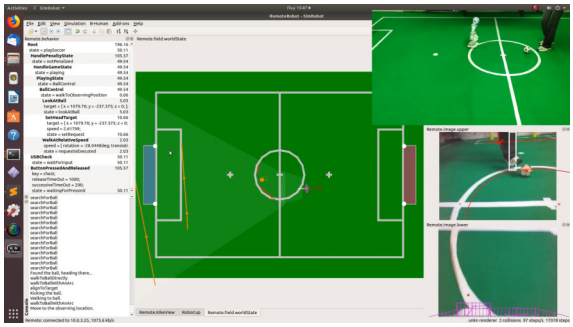
Figure 5.26: Ball Control: Simulation Results of the proper positioning.



(a)



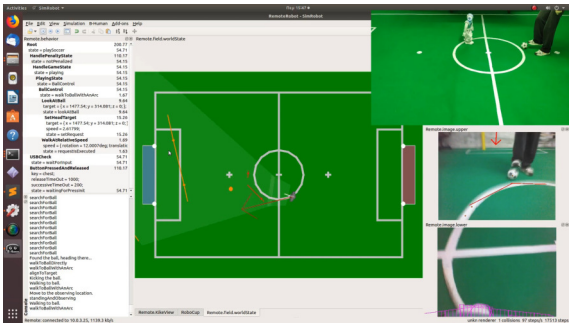
(b)



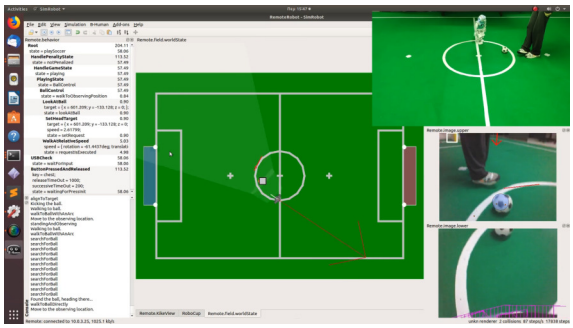
(c)

(d)

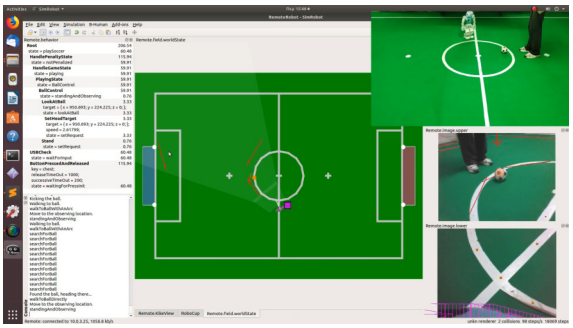
(e)



(f)



(g)



(h)

Figure 5.27: Ball Control: Lab Results of the proper positioning.

Chapter 6

Conclusions

6.1 Discussion

We understood a good portion of the code and the tools included and we believe that we can provide enough information for the implementation of more complex behaviors, even though we might have neither managed to implement optimal nor extremely competitive behaviors. In addition, we believe that the current framework could be utilized beyond the spectrum of soccer.

Simulation Vs Reality

A variety of conclusions were drawn before, during and after the behavior implementations. First of all, it was important to understand how different a behavior can be depending on the environment. The environment of the simulator is close to ideal and while it can help up understand how a behavior should work, does not show us how it does work on the robot. An important example is the impact walking has on footage of the Nao's cameras. But it is not only the difference between the simulation and reality but also the differences between different machines running the same simulation. Since we started experimenting on SimRobot we didn't fully understand how impactful that may be.

The speed of the execution of the simulation depends on the specs of the hosting machine. If the machine is not as powerful the drop in FPS[1] can be devastating thus the

[1]Frames per second.

developer would be unable to properly understand the behavior. For instance, a behavior may seem very slow in around 20 fps and when executed on an actual robot could cause damage on the joints. Also it seemed to cause issues on the execution circle although these issues were too sparse.

Having a too powerful machine on the other hand, can also lead to inconsistencies because the simulation tends to run slightly faster than the actual robot. Something that can lead to the usage of value, specially speed values that can cause damage to the actual robot when used in the lab.

While testing on the robot itself seems the only option, that could also lead to potential damage since we aren't exactly sure how something behaves until we try it on the actual robot. So the best plan of developing and testing behaviors is the combination of both simulation and real robot with extra caution when trying things on the Nao itself.

Another drawn conclusion was that the best way to test a behavior on an actual robot is to set a speed around half or a bit lower of the value range. Setting it to the lowest possible might also have a negative impact because some actions require momentum in order to be executed correctly. With the use of an average value, the damage to the hardware maybe be curtailed.

In addition, since the behaviors tend to run a bit slower on the robot, state timers and the `timeSinceBallWasSeen` timer needs to be adjusted accordingly. For instance, setting the condition that checks the timer of the `timeSinceBallWasSeen` too low in order to have a faster response of the robot when the ball is not seen will work on the simulation but it will result on the actual robot searching for the ball constantly. This occurs because the robot wont be able to keep an accurate track of the ball even while walking towards it because some parts of the walking animation cause distortion of the camera image and as a result not seen the ball for a good amount of milliseconds.

6.2 Future Work

The work done in this thesis is the initial steps and while the project provides us with all the necessary tools to work with different strategies and scenarios we didn't dive deeper into that field due to the lack of hardware and to reduce the complexity. Based on the work done so far we believe that any future work aiming for more complicated behaviors it would be more efficient if done in the order presented in this section.

6.2.1 The Next Step

Code re-factoring in general

A lot of positioning management should be move to upper levels in the behavior functionality, in an additional behavior layer. Setting the boundaries, calculating the path our player should take in order to reach its destination, the appropriate kicking foot for each kick as well as the proper alignment before each kick. All of these tasks are currently executed in the action section of the behaviors with the use of extensive conditioning even though the behaviors themselves are not that sophisticated.

Adding an extra module handling the above will also solve some dependencies and make the project architecture clearer that was also partially intentionally tangled in order for the implemented behaviors to have access to some modules such as field dimensions. This will also provide working ground for a more sophisticated localize, field scan or even ball search. Those states, instead of a predefined movements, they could be guided by a module on a higher level and pass the values required for movement straight to the behavior and then the required engine. That would also releaf the behavior control from calculations that might delay the robot's actions.

Thus we believe that one of the important next steps in this implementation would be the code re-factoring and integration of such a module for soccer and non-soccer behaviors.

For implemented soccer behaviors

Focusing more on the soccer behaviors after the code re-factoring there could be a few improvements. For the implemented behaviors themselves there would be a few improvements.

The simple striker should be able to identify multiple obstacles and handle the dibbling through them.

The simple defender should be able to block an incoming pass intercepting an attack and also moving to the ball while blocking an opponent. The latter would require the use of other team members that would indicate the coordinates of the blocked opponent.

The simple goalkeeper should be able to execute a blocking move in case of a threatening shot.

Another future step would be to figure out how to assign roles to multiple robots. This step does not necessary require multiple real robots. SimRobot could be a temporary solution into trying out simultaneously different behaviors but that might required a strategy implementation instead of deploying different builds to different robots based on the wanted behavior.

When more robots are part of the team they should rely more to the `TeamBallModel` and not so much to their own `BallModel` since its a combination of the others robots perception. For that to happen, the implementation must be adjusted as well because the local and global coordinates vary.

For that to work properly, we need the game controller because if the robot is not communicating with a game controller the state of the game is not updated correctly. If that is the case the `theTeamBallModel` variable gets constantly reseted and thus unusable. More information of how to temporary bypass this problem is included in the appendix.

Another important feature to be added is the game controller. This requires to find a way to make the provided game controller work and also connect it to the network that our robots are playing. When that takes place then the implemented behaviors need to be broken down to small ones. For instance, the positioning the goalkeeper executes should take place at the set game state. In addition, the code responsible for the handling the Kick off needs to be re-enabled.

Static values such as positions and targets should be used as a proportion of the field dimensions. These value should also be provided by a higher module and the roles should not have direct access to features such as `fieldDimentions` nor make calculations on them. That would make the behaviors cleaner, universal and maybe even faster.

Last but not least, the `PathPlannerProvider.cpp` provided must somehow be checked, comprehended and included allowing our robots to avoid obstacles real time.

6.3 Lessons

During the course of this thesis, we became familiar with a different type of project architecture that is used in projects of this scale. A variety of configuration files are gathered and set easily accessible in the higher levels of the architecture. There is also a different setup files depending on the operating system with the accompany scripts

for the generation of the proper project build for different IDEs[1] that led to a better realization of the architecture on multiple levels.

We also became familiar with navigating and searching useful information with the use of the command line in such a project where the internet and a search engine wont have all the answers. In addition, we gained experience and a better understanding of operating systems, both Linux and Windows, such as handling permission issue. Since we were alternating between the two OS we also practice the usage of Git.

Lastly, we now have a better understanding of how to proper set up and handle different networks.

[1] Integrated Development Environments

Bibliography

[1] Kitano, H., Asada, M., Kuniyoshi, Y., Noda, I., Osawa, E., Matsubara, H.: RoboCup: A challenge problem for AI. AI Magazine **18**(1) (1997) 73–85 https://www.robocup.org. 5

[2] Röfer, T., Laue, T., Bülter, Y., Krause, D., Kuball, J., Poppinga, A.M.B., Prinzler, M., Post, L., Röhrig, E., Schröder, R., Thielke, F.: Team Report and Code Release 2017. Technical report, University of Bremen and the German Research Center for Artificial Intelligence (DFKI) (2017) https://github.com/bhuman/BHumanCodeRelease/blob/coderelease2017/CodeRelease2017.pdf. 20, 21, 27, 31, 131, 134, 135

[3] Röfer, T., Laue, T., Hasselbring, A., Heyen, J., Poppinga, B., Reichenberg, P., Röhrig, E., Thielke, F.: Team Report and Code Release 2018. Technical report, University of Bremen and the German Research Center for Artificial Intelligence (DFKI) (2018) https://github.com/bhuman/BHumanCodeRelease/blob/master/CodeRelease2018.pdf. 20, 27, 34, 121, 129, 133, 134

[4] Kermani, R.R.: Model-based Design, Simulation and Automatic Code Generation For Embedded Systems and Robotic Applications. Technical report, Arizona State University (2013) http://www.public.asu.edu/~gfaineko/pub/thesis/kermani_2013_ms_thesis.pdf, https://subversion.assembla.com/svn/simulink-stateflow-interface-for-nao/. 26

[5] Brohn, T., Ditzel, S., Fürtig, D.I.A., Glaser, L., Hess, T., Hammer, H.J., Knorr, E., Rinfreschi, K., Schön, T., Siegl, J.M., Steiner, S., Weiglhofer, F., Wörner, P.: RoboCup SPL Team at Goethe University Frankfurt Teamreport 2018. Technical

report, Goethe University (2018) https://www.jrl.cs.uni-frankfurt.de/web/wp-content/uploads/2018/01/teamreport18-bembelbots.pdf. 27

[6] Mellmann, H., Schlotter, B., Kaden, S., Strobel, P., Krause, T., Couque-Castelnovo, E., Ritter, C.N., Hübner, T., Tofangchi, S.: Berlin United - Nao Team Humboldt Team Report 2018. Technical report, Humboldt University Berlin (2019) https://www2.informatik.hu-berlin.de/~naoth/docs/publications/technical/naoth-report18.pdf. 27

[7] Aizawa, Y., Hidaka, K., Mori, N., Ohkusu, K., Takahashi, K., Uemura, Y., Chiba, M., Hayashi, K., Ito, K., Nagami, T., Shimizu, Y., Yamada, Y., Hosokawa, K., Ito, S., Kuboya, T., Ohta, G., Tachi, T., Tanabe, K., Tsubokura, K., Suzuki, T., Kobayashi, K.: Camellia Dragons Team Report 2017. Technical report, Aichi Prefectural University (2018) https://github.com/CamelliaDragons/CamelliaDragonsCodeRelease/blob/master/CamelliaDragonsTeamReport2017rev2.pdf. 27

[8] Akin, H.L., Aşik, O., Görer, B., Erdem, A., Irfan, B.: Cerberus'14 Team Report. Technical report, Bŏgaziçi University (2015) http://robot.cmpe.boun.edu.tr/~cerberus/wiki/lib/exe/fetch.php/wiki:cerberus14report.pdf. 27

[9] Morales, M., Hayet, J.B., Esteves, C., Anaya, R.: Cuauhpipiltin 2012: Standard Platform League Team Description Paper. Technical report, ITAM, CIMAT, UNAM, and DEMAT-UG (2015) http://www.robotica.itam.mx/spl/tdp_spl_ek_robocup2012.pdf. 28

[10] Heßler, A., Xu, Y., Tuguldur, E.O., Berger, M.: Team Description 2014. Technical report, Technical University of Berlin http://dainamite.github.io/public/publication/dainamite_tdp_2014_final.pdf. 28

[11] Kronemeijer, P., Lagrand, C., Negrijn, S., van der Meer, M., van der Wal, D., Nzuanzu, J., Hesselink, R., Zwerink, W., Gupta, A.R.K.N., Visser, A.: Team Qualification Document for RoboCup 2019. Technical report, University of Amsterdam (2019) https://www.dutchnaoteam.nl/wp-content/uploads/2019/01/DNT_TeamQualificationDocument2019.pdf. 28

[12] Riebesel, N., Hasselbring, A., Peters, L., Poppinga, F.: Team Research Report 2016. Technical report, Hamburg University of Technology https://hulks.de/_files/TRR_2016.pdf. 28

[13] Kargas, N., Kofinas, N., Michelioudakis, E., Pavlakis, N., Piperakis, S., Spanoudakis, N.I., Lagoudakis, M.G.: Kouretes 2013 SPL Team Description Paper. Technical report, Technical University of Crete http://www.intelligence.tuc.gr/kouretes/docs/2013-kouretes-tdp.pdf. 28

[14] Paraschos, A., Spanoudakis, N.I., Lagoudakis, M.G.: Model-driven behavior specification for robotic teams. In: Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2012). Volume 1., Valencia, Spain (2012) 171–178 28

[15] Spanoudakis, N.: The Agent Systems Engineering Methodology ASEME. Phd thesis, Paris Descartes University, Paris (2009) 28

[16] Topalidou-Kyniazopoulou, A., Spanoudakis, N.I., Lagoudakis, M.G.: A CASE Tool for Robot Behavior Development. Technical report, Technical University of Crete https://users.isc.tuc.gr/~nispanoudakis/resources/RCS-2012-TopalidouSpanoudakisLagoudakis.pdf. 28

[17] Wooldridge, M., Jennings, N.R., Kinny, D.: The Gaia Methodology for Agent-Oriented Analysis and Design. Autonomous Agents and Multi-Agent Systems **3**(3) (2000) 285–312 28

[18] Estivill-Castro, V.: Standard Platform League Team Description. Technical report, Griffith University and Universitat Pompeu Fabra http://www.mipal.net.au/Publications/TDpaperMIPAL.pdf. 29

[19] Hofmann, M., Schwarz, I., Urbann, O., Larisch, A.: Team Report 2018. Technical report, Robotics Research Institute https://github.com/NaoDevils/CodeRelease/blob/master/TeamReport2018.pdf. 29

[20] Shih, C.S., Lin, K.L., Chiang, S.A., Hu, T.H., Fu, W.K., Hu, C., Chen, H.Y., Cheng, Y.Y., Chao, T.W.: NTU RoboPAL Team Report 2015. Technical

report, National Taiwan University (2015) https://drive.google.com/file/d/0B1d3UWuPZfdyVjlKRU1iTVpqUGc/view. 29

[21] Suriani, V., Manoni, T., Giambattista, V.D., Cecchini, M., Bloisi, D.D., Nardi, D.: Team Description Paper 2018. Technical report, Sapienza University of Rome http://www.diag.uniroma1.it/~spqr/reports/SPQR-TDP2018.pdf. 29

[22] Saifullah: Team Report - 2019. Technical report, RISE, SMME and NUST (2019) https://docs.google.com/document/d/1H9NFfVEYYqDxaf2ef4AEArm_PaG9ix4DcYhPoQEpLxI/edit. 29

[23] : Team description paper & research report 2018. Technical report, Tongji University (2019) https://github.com/TJArk-Robotics/TJArK-Vision/blob/master/TJArkTeamResearchReport2018.pdf. 29

[24] Brameld, K., Hamersley, F., Jones, E., Kaur, T., Li, L., Lu, W., Pagnucco, M., Sammut, C., Sheh, Q., Schmidt, P., Wiley, T., Wondo, A., Yang, K.: RoboCup SPL 2018 rUNSWift Team Paper. Technical report, University of New South Wales Sydney http://cgi.cse.unsw.edu.au/~robocup/2018/TeamPaper2018.pdf. 29

[25] Deng, X., Pak, D., La, F., Lee, D.D.: The UPennalizers RoboCup Standard Platform League Team Description Paper 2018. Technical report, University of Pennsylvania https://fling.seas.upenn.edu/~robocup/files/2018Report.pdf. 30

[26] Committee, R.T.: RoboCup Standard Platform League (NAO) Rule Book. RoboCup Technical Committee (2018) https://spl.robocup.org/wp-content/uploads/downloads/Rules2018.pdf. 32, 39

[27] Röfer, T.: CABSL C-based Agent Behavior Specification Language. Technical report, University of Bremen (2018) https://www.b-human.de/downloads/publications/2018/CABSL.pdf. 40

[28] Loetzsch, M., Risler, M., Jungel, M.: XABSL - A Pragmatic Approach to Behavior Engineering. Technical report, Technische Universität Darmstadt and Humboldt-Universität zu Berlin http://martin-loetzsch.de/publications/loetzsch06xabsl.pdf. 40

Appendix A

User Guide

Requirements

In order to execute the code the reader will need:

- Our code [Link - GitHub repository.].
- A relative powerful machine in case you want to run simulations.
- A Nao v.5 with B-Human installed[1].

A.1 How to run the existing code

In order to execute the existing code you need to navigate to the `\Make\` directory found in the upper levels of this project and depending on the operating system, execute one of the following options. For Linux, navigate to `\Make\Linux\` and execute in a terminal `make all -j`. For Windows it would be advised to use VStudio and that requires to generate the appropriate project from the directory `\Make\VS2017\` with the use of the generate script located in the mentioned directory. For windows it should also be noted that you will need Cygwin as stated in the B-Human code release [3] in Chapter 2, Getting Started. As a build configuration we are using develop. After the compilation is finished we navigate to `\Make\Linux\` in order to use another on of the scripts, copyfiles in order to copy the compiled code to the robot. You can use `./copyfiles` –help for more information regards the use of this command.

[1] In case of not having it installed refer to Section B

A.1.1 Alternating between behaviors, scenes and locations

How to switch between the roles/behaviors

In order to alternate between the existing behaviors with the game states in this implementation you must navigate to `HandlePlayingState.h` located at `\Src\Modules\BehaviorControl\BehaviorControl\Options\GameControl\` and remove the comments from the behavior you wish to execute. This way, in order to execute the behavior, you will need to penalize and not penalize the robot manually when working with a real robot. For simulation you need to write in the console `gc playing` in order to set up the game state to playing.

If you wish to bypass the game flow you can do so, only for the `BallControl` for the current implementation, by going to `Soccer.cpp` at `\Src\Modules\BehaviorControl\BehaviorControl\` and call the appropriate option.

How to change location/scenario

In this project we are able to add and alternate between locations.

In order for the localization to work properly, the dimensions of the location must be set accordingly since the map is considered known and no additional mapping algorithm is being executed. The same procedure must be done for the scenes in SimRobot since the Nao comprehends its surroundings and localizes based on the field dimensions and landmarks, such as ground lines and goalposts. In our case, the dimensions used in SimRobot are different from the dimensions of the field we have in the lab. As a result we need to change the location of the execution based on the robot we run it, real or simulated robot. We have added an extra location called Lab with different dimensions. One way to switch to that location is to set the `location` variable of the `settings.cfg` located at `\Config\` to the location folder we have created and added to `\Config\Locations\`, in our case Lab. Then we compile and copy the files to our robot or run the simulation.

Locations have multiple variables defining them, from field dimensions to camera calibrations and color settings. The same principle is followed for the scenarios as well, where we can set different behavior and motion parameters.

How to start SimRobot and change scenes

In order to launch SimRobot you need to navigate to the folder with the files created from the projects compilation. Depending on the operating system the executable of SimRobot can be found at the corresponding directory, for Windows is at `\Build\Windows\SimRobot\Develop\`. All the scenes can be located at `\Config\Scenes\`. This is also the folder you will need to navigate when you want to alternate and select scenes after launching SimRobot. When you launch SimRobot you need to select a scene for any simulation or robot monitor to take place. If you had selected a scene in the past it will be included in a recent menu at the arrow next to the folder icon on the top left. `RemoteRobot.ros2` is the scene used for remote access and it will attempt to reconnect to robot IP used in the past. Some scenes can be faster on execution than others and that depends on the objects included in the scene.

A.2 Getting Started with Behavior Development

A.2.1 Commands

Useful commands regardless the operating system, since Cygwin is also installed on Windows.

- `grep`
- `chmod`
- `make -j` where you are able to assign jobs in order to speed up the compile.

It also advised to use an IDE in order to assist with locating the files and the references. The VStudio project seemed to be really well organized and helped speed up going though classes and option files trying to locate the initialization of an object in order to understand its usage.

We can build each module seperatally by running make and the name of the module, for instance `make SimRobot` or `make bush`.

How to search around

We usually use grep in order to search and navigate around the project by typing in a terming:

```
grep -i -r "what we search" "directory to search in"
```

and we can also do the same for files and add type of files we would like to exclude from our search, such as binary files. For instance by using `grep -rI` you can exclude binary files. Or you can `--include="*.example"` and `--exclude="*.example"` specific file formats to adjust the search in your needs.

A.2.2 SimRobot Quick Guide

Setting up the required scenes

In order to have all the available information easily accessible at all times we must utilize a couple of SimRobot's views. During the implementation of the behaviors included in this thesis we used the following views enabled, usually in the arrangement shown in most of the figures in this thesis.

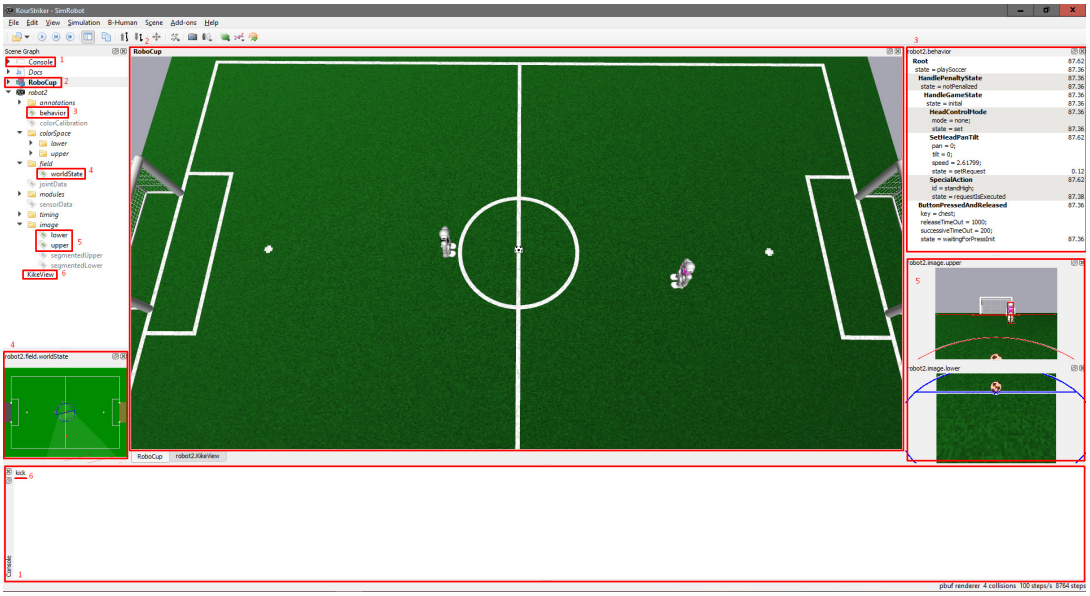
- behavior view
- worldState view
- upper and lower camera views
- console view
- kike view

In a more detailed Figure A.1 we can see where are the above view located in order to enable them.

A.2.3 Quick Implementation Guide

In this section we are going to go through the absolutely basic steps that are needed in order to write, compile and run a behavior based on our experience, given that the installation of B-Human on a computer for the SimRobot and a Nao is already done[1].

[1] If not, please refer to Section B



(a)

Figure A.1: SimRobot feedback views. We can see how to enable each of the shown views. The only tricky one is KikeView where you need to type kick in terminal while the simulation is running in order for the KikeView to appear on the left.

How to control Nao

You can control Nao's movements with the use of two motion requests that were also mentioned in the previous chapters, `walkAtRelativeSpeed` and `walkToTarget`, although for the latter you will need to use variables that are relative to the robot. These requests are called inside the action section of the state. In the same state, in the transition section, we add the required condition that checks the results of the motion request. As a general rule, we don't set the acceptable angle or any other variable we check to a specific number because that approach could lead to deadlocks since there are always smalls overshoots or undershoots both to the movement of the robot and and to the localization. Instead we are checking a range of values. That is true in general when creating behaviors, allowing zero error on values can lead to deadlocks appearing as oscillations. A simple example would be to ask the robot to move to X location. Its highly unlikely that the robot will manage to match its x to be exactly equal to X. In addition, there is also error in both X and x variables. As a result, it will start going forwards to an overshoot and backward to

an undershoot, specially if the speed of the robot is not adjusted based on the distance. You can control the Nao's head by defined the `HeadControlMode` you wish it to execute in every state along side the rest motion controls in the action section of a state.

How to add new roles/behaviors

In order for a role or any other configuration to work, we need to add it in the sub-folders of the following directory `\Src\odules\BehaviorControl\BehaviorControl\Options\Roles\` and also include the path of the newly added folder in the `Options.h` that is located at `\Src\Modules\BehaviorControl\BehaviorControl\.`

For each behavior, is mandatory to have an initial state or else it will cause a crash when executed. After adding the required files you must include them in one of the game states. Until a game controller is added to the network the best option would be the `PlayingState.h` located at:
`\Src\Modules\BehaviorControl\BehaviorControl\Options\GameControl\`
You can control the game flow to transition to this state either by typing in the terminal `gc playing` for the simulation or by cycling manually through the penalization of the actual robot by pressing the button on its chest.

How to add new head control modes

If we want to add new head control modes we need to start by creating the new head mode file. The head control modes are in the following directory:
`\Src\Modules\BehaviorControl\BehaviorControl\Options\HeadControl\`

We create the new file giving it the same name as the mode we wish to implement. That name will also go as the first argument of the option. We can use one of the other modes as a reference. In this mode we are using different head motion request located at the `\HeadMotionRequest\` folder in `\Src\Modules\BehaviorControl\BehaviorControl\Options\Output\`. We also need to include our newly create mode into a few locations. Firstly, at the `Option.h` located at `\Src\Modules\BehaviorControl\BehaviorControl\` where we need to include the location of the file. Then, at the `HeadControlMode.h` located at `\Src\Modules\BehaviorControl\BehaviorControl\Tools\` where we need to add it in the `ENUM`. Lastly, at the `HeadControl2018.cpp` where we also need to include the location of the file and at `HeadControl2018.h`. Both files are

located at `\Src\Modules\BehaviorControl\BehaviorControl\HeadControl2018\`. In the `HeadControl2018.cpp` file we need to add a case for our head control mode. The head motion `HeadMotionRequest::targetOnGroundMode` seems that is not included when we look at the `CameraControlEngine.cpp` located at `\Src\Modules\BehaviorControl\CameraControlEngine\` and need implementation for the coordinate transformation.

How to create new locations

We need to add the new location folder at the `\Config\Location\` and then change the location parameter in the `settings.cfg` at `\Config\`. The best would be to copy one of the existing ones and alter it as we please.

FieldDimensions

Field dimension are bound into locations. For instance, there is the default location with the field dimensions of a field used in RoboCup 2016 and also complies with simulated field in SimRobot. The field in our lab is a much older version with different dimensions as well as goalposts. All these differences must be specified in the field dimensions where there is a variable for every single line and distance on the field before the initial calibration as well B.

Switching between the two is really important and if the dimensions are not set up correctly the perception module will conclude in wrong values and even the localization itself will be wrong.

After creating the files required for the different locations, we can switch between them in various ways.
- Either by manually changing the configuration file before or after building and uploading the build code to Nao.
- Via the B-Human bush, an application that is able to manipulate of all the robots connected to it the default network, scenario and location.
- Via command line while being connected with Nao with ssh and changing the configuration file. Since it is a configuration file its not required to be build with the whole solution in order to be used.

How to add new kicks using InWalkKicks

In order to create different types InWalkKicks we need to add our new inWalkKick to `\Src\Representations\Configuration\WalkKicks.h` and then included at `WalkKicks.h` located at `\Config\WalkKicks`. Then we can call it by using the InWalkKick as seen in the Listing A.1

Listing A.1: InWalkKick example

```
\* How to call the inWalkKick myinwalkkick *\

InWalkKick(WalkKickVariant(WalkKicks::myinwalkkick, Legs::left),
    Pose2f(theBallModel.estimate.position.angle(),theBallModel.estimate.position.x()
    - 100, theBallModel.estimate.position.y() - 50));
```

How to add new kicks using KickEngine

For the Kick Engine kicks we have already added a basic motion request, `ForwardKick.h` that can be used as a reference. In the future you might need to make a new motion request. In order to do so you must add the new motion request in `\Src\Modules\BehaviorControl\BehaviorControl\Options\Output\MotionRequest` and also include it in `Options.h`.

We can create new kick files with the use of KikeView. After creating the new kick we must add the `KickMotionId` in `KickRequest.h` located at `\Src\Representations\MotionControl\`. The id of the kick is the same with its name. The next step would be to also create the appropriate motion request that will request the execution of the kick in `\Src\Modules\BehaviorControl\BehaviorControl\Options\Output\MotionRequest`. After creating the motion request you now have to call it in the action section of the state you wish to execute it. When creating kicks in kike view, there are a few things to keep in mind. When you have the Kike View selected, the icons on the top left of SimRobot change allowing you to import .kmc files along with other functions. It would be advised to have as a reference the `kickForward.kmc` and the window as big as possible so you wont miss any important information but in the same time to avoid hiding that menu. It is also important to note that when you create your own kick do not leave the `COM Balancer X` and `COM Balancer Y` to zero cause it will cause crashes. While it might

sound obvious, since the view can appear really small the names and value cut out and can be easily missed.

How to create a local network

SPL network is special since the IP of the robot indicates information regards its number and team. For instance Ermis, our Nao is the number 25 of team 3. So its IP address is 10.0.3.25 and for LAN 192.168.3.25. In order to be able to connect to this robot you need to set up the appropriate network to communicate with it as a team member.

In order to be able to connect to Nao with the new IP you will need to create two different networks, one wired and one wireless(if the computer has a wireless adapter) at the computer that will have access to the robot. An example of wired configuration can be found in https://github.com/AnonKour/LabTutorials/tree/master/HowToS/ConnectToNewerNaoBHuman. In the same way, we set up the wireless network. If you setup you robot with a different team number you need to adjust the IP accordingly.

For the wireless communication you will also need to set up an access point, set the `IP Address` to 10.0.3.1 with `Subnet Mask`: 255.255.0.0 and `DHCP` and `DHCP Server` enabled. We also set the `IP Address Pool` to cover the IPs of you robots. In our case, we use a pool of 10.0.3.2 - 10.0.3.99 `Default Gateway` and `Default Domain` to 10.0.3.1.

How to setup the networks on Nao

In 2.4.3 Creating Robot Configuration Files for a NAO you setup the required configuration files that also store the IP for wire and wireless for you robot. Be extra careful not to have the same IP for wireless and wired because Nao will refuse to connect wirelessly.

In the 2.4.4 Managing Wireless Configurations of the B-Human code release [3] you will need to configure the required wired and wireless supplicants based on the setup of your access point.

There are examples of that format located at `\Install\Network\Profiles\`.
Some useful links that could help you set up the wpa_supplicant properly in addition to the examples:

- https://wiki.netbsd.org/tutorials/how_to_use_wpa_supplicant/
- https://linux.die.net/man/5/wpa_supplicant.conf

- Cnet guide.[1]

How to connect to Nao/Wired/Wireless

There are two ways to connect to Nao. Via WIFI while being in the same wireless network and via Ethernet. Once you are in the the same network you could use either ssh with the following format nao@*"and Nao's IP"* or a script located at `\Make\Linux\` folder under to operating system or simply in the Common folder.

How to add behaviors that bypass the game flow

You can call it from `\Src\Modules\BehaviorControl\BehaviorControl\Soccer.cpp` and you can add your code at `\Src\Modules\BehaviorControl\BehaviorControl\Options\Soccer.h` as another option. While bypassing the game flow is possible, it could effect other modules such as perception and localization.

Be careful when naming cause the option should not have the same name as one of its states.

How to access the bumpers and touch sensors

In order to see how you can access the bumpers, the buttons and the touch sensors of Nao you can check the `KeyStates.h` located at `\Src\Representations\Infrastructure\SensorData\`. You can see an example of how to use those `KeyStates` in `Soccer.h` located at `\Src/Modules\BehaviorControl\BehaviorControl\` where the chest button is being used.

How to copy the compiled code to Nao

In order to copy files to the robot we use `copyfiles` which is located at `\Make\Linux\` and we put the robot's IP as well as the options we want. We can see a full list of the options if we use `./copyfiles -h`

[1] https://www.cnet.com/how-to/home-networking-explained-part-5-setting-up-a-home-router/

How to run B-Human bush

If you have executed `make all` then you have also compiled the The B-Human User Shell (bush), a tool described in Section 10.2 B-Human User Shell of B-Human code release of 2017 [2]. It can be found in `\Build\Linux\bush\Develop\`.

How to connect Nao via SimRobot

When starting a scene, if a robot has connected in the past with SimRobot the tool attempts to connect to that IP. If the B-Human module is running at the robot and everything else is setup properly it will establish a remote connection automatically. We can check the status of naoqi and the bhuman module by typing `status` while connected to the robot.

Things to keep in mind

Be careful when using targets with `Vector3f` cause the variable order is different that `Pose2f` used in walking motion requests. The former uses a variable order of x, y and angle while the latter of angle, x and y.
As stated in the code release [2]:

The `TeamBallModel` is the perception that was created by the images of all the robots in order to find the global position of the ball and hence to be able to locate it on the field and not based on the relative position of the robot.

The obstacle detection does not distinguish between different types of obstacles, i. e. robots, goal post, and referees. In fact, it does not detect referees wearing black trousers at all, because black is considered to be a possible color of the field. All obstacles found are treated as robots. Therefore, it is tried to detect their jersey color.

All the console commands are located in chapter 10.1.6 Console Commands in B-Human code release [2]

You might need to setup different values for timer that checks how much time has passed since last seen the ball. Different motions can distort the camera image causing a momentary loss of the ball even if its in the robot's field of view.

When working with a real robot make sure to clean up the log files regularly cause they will end up allocating all the available space. It will cause eventually a `rsync`

`write error` when you will try to copy the compiled code to Nao via `./copyfiles`. Check `downloadLogs` script in `\Make\Linux\` for more information.

Appendix B

Installing B-Human Code Release

This section complements the B-Human section in code release 2018 [3], Getting Started. Before following the steps mentioned in section 2.4 Setting Up the NAO, there are a few things you should keep in mind.

For the atom system image and the flasher mentioned you will need an account with a **registered** robot is required (https://community.ald.softbankrobotics.com). In order to find those files, after signing in, you must navigate to Resources in the top right and then Software on the tab menu at the center of the screen. Then select English and under My Robots select Nao. You will find the required files in the list.

Take also into consideration that flashing lasted for a 2014 Nao v.5 around 20 minutes. Until the first boot in needed 26 minutes, so be patient.

Also every time we tried the procedure got halted in the register page of the robot step regardless of the account we used to register it. So when you reach that step, restart Nao[1] in order to get the settings from the cloud but make sure that you passed the step where you set up the password. If you restart the Nao before setting up a password or if you backtrack the steps it can end up without a password and it seems in order for that to be resolved it need to be reformatted since there is no way we know of to connect to it. After the restart when you enter the IP of the robot in a browser it will automatically fetch the correct credentials and allow you to finish the setup.

After the B-Human installation, while we are used to Nao greeting us, all those procedures with the installation of B-Human are being disabled. B-Human installation

[1] If it has been already registered in the past.

also lasts a few minutes.

How to setup Nao to work with B-Human

The steps required to setup Nao are in the section 2.4 Setting Up the Nao of the B-Human code release 2018 [3]. Be extra careful not to have the same IP for wireless and wired because Nao will not be able to connect wirelessly to the network. For creating the wired and wireless files see A.2.3.

How to calibrate Nao

The procedure we must follow in order to calibrate Nao is stated in section 2.8 Calibrating the Robots of B-Human code release 2018 [3]. The commands of the chapter must be typed in SimRobot's console. You can see how to setup the required scenes in Section A.2.2 of the appendix. You can also use the `Tab` button on your keyboard to assist with the auto completion and cycling between options. Be careful to use the correct field dimensions regards the robot's coordinates. All the console commands are located in chapter 10.1.6 Console Commands in B-Human code release 2017 [2]. After further research, it would seem that when we calibrate our Nao we skipped an important step since we couldn't properly understand how properly run the calibrations. From what we understood the required steps are the following although we did not have the chance to confirm that:

- First we need to execute the stand and calibration stand.(Not sure if you need both.)
- Then we need to make sure that the legs are properly aligned as mentioned in the code release and do any manual adjustments to the robot stance as necessary in order to correct its actual posture. A tape measure or ruler is required.
- Then we call the calibrator in order to extract the values of the manual adjustments we made.
- Then save the representation.
- Redeploy Nao(either meaning copy files or install robot?) and restart B-Human.

In Listing B.1 we can see the console commands that are required in order to execute the steps mentioned. After that, in the same principle, proceed with the 2.8.3 Camera

Calibration. The `Join` Calibrator seen in the previous listing is located in `\Config\Scenes\` and as we can see in Listing B.2, it handles the `CalibrationStand`. For the rest of the calibrations follow the instructions on the code release 2017 [2] Chapter 2.8.3 Camera Calibration. After that there is Chapter 2.8.4 Color Calibration which we did not run.

Listing B.1: B-Human console

```
/* We type in the console the command. */

get representation:MotionRequest

/* And we get as a result a pre-configured command that we need to alter and execute:
    */

set representation:MotionRequest motion = specialAction; specialActionRequest = {
    specialAction = standHigh; mirror = false; }; walkRequest = { mode =
    absoluteSpeedMode; speed = { rotation = 0deg; translation = { x = 0; y = 0; }; };
    target = { rotation = 0deg; translation = { x = 0; y = 0; }; }; walkKickRequest =
    { kickType = none; kickLeg = left; }; }; kickRequest = { kickMotionType = none;
    mirror = false; armsBackFix = false; autoProceed = false; boost = false;
    dynPoints = []; };

/* We need to alter the command and set motion = stand. */

set representation:MotionRequest motion = stand; specialActionRequest = {
    specialAction = standHigh; mirror = false; }; walkRequest = { mode =
    absoluteSpeedMode; speed = { rotation = 0deg; translation = { x = 0; y = 0; }; };
    target = { rotation = 0deg; translation = { x = 0; y = 0; }; }; walkKickRequest =
    { kickType = none; kickLeg = left; }; }; kickRequest = { kickMotionType = none;
    mirror = false; armsBackFix = false; autoProceed = false; boost = false;
    dynPoints = []; };

/* The robot will change its stance accordingly. In order to call the
    CalibrationStand you call the Joint Calibrator: */

call Calibrators/Joint

/* From what we understood this is the part where we need to check the distance
    between the Nao's feet in order to make sure its 10cm as stated in the code
    release. Then these values need to be given to JointCalibration via typing: */

get representation:JointCalibration

/* This should give us a different result than the following which is the default.*/

set representation:JointCalibration offsets = { headYaw = 0deg; headPitch = 0deg;
    lShoulderPitch = 0deg; lShoulderRoll = 0deg; lElbowYaw = 0deg; lElbowRoll = 0deg;
```

```
    lWristYaw = 0deg; lHand = 0deg; rShoulderPitch = 0deg; rShoulderRoll = 0deg;
    rElbowYaw = 0deg; rElbowRoll = 0deg; rWristYaw = 0deg; rHand = 0deg; lHipYawPitch
    = 0deg; lHipRoll = 0deg; lHipPitch = 0deg; lKneePitch = 0deg; lAnklePitch = 0deg;
    lAnkleRoll = 0deg; rHipYawPitch = 0deg; rHipRoll = 0deg; rHipPitch = 0deg;
    rKneePitch = 0deg; rAnklePitch = 0deg; rAnkleRoll = 0deg; };

/* Then save the representation. */

save representation : JointCalibration

/* Redeploy Nao(either meaning copy files or install robot?) and restart B-Human */
```

Listing B.2: B-Human Calibrators - How they execute CalibrationStand

```
mr JointCalibration JointCalibrator
dr module:JointCalibrator:init
mr FallEngineOutput default

mr StandArmRequest CalibrationStand
mr StandLegRequest CalibrationStand
set module:CalibrationStand:jointCalibrationMode true;
mr WalkingEngineOutput default
dr module:CalibrationStand:height

echo dr module:JointCalibrator:reset
echo dr module:JointCalibrator:reload
echo dr module:JointCalibrator:capture
echo save representation:JointCalibration

get module:JointCalibrator:offsets
```