

The Behaviour-Oriented Design of an Artificial Football Team

Thomas Hyde

Bachelor of Science in Computer Science
with Mathematics with Honours

The University of Bath

May 2012

The Behaviour-Oriented Design of an Artificial Football Team

submitted by Thomas Hyde

COPYRIGHT

Attention is drawn to the fact that copyright of this dissertation rests with its author. The Intellectual Property Rights of the products produced as part of the project belong to the University of Bath (see <http://www.bath.ac.uk/ordinances/#intelprop>). This copy of the dissertation has been supplied on condition that anyone who consults it is understood to recognise that its copyright rests with its author and that no quotation from the dissertation and no information derived from it may be published without the prior written consent of the author.

Declaration

This dissertation is submitted to the University of Bath in accordance with the requirements of the degree of Bachelor of Science in the Department of Computer Science. No portion of the work in this dissertation has been submitted in support of an application for any other degree or qualification of this or any other university or institution of learning. Except where specifically acknowledged, it is the work of the author.

Signed:

This dissertation may be made available for consultation within the University Library and may be photocopied or lent to other libraries for the purposes of consultation.

Signed:

Abstract

Behaviour-Oriented Design is a development methodology for the creation of complex intelligent agents. Agents created using Behaviour-Oriented Design are modular, and use hierarchical reactive plans. I apply the methodology to the creation of an artificial intelligence football team for use with the RoboCup 2D Simulation Soccer Server. I use this experience to critically analyse Behaviour-Oriented Design, and assess its suitability for such a purpose.

Contents

List of Figures	4
List of Tables	5
1 Introduction	7
1.1 Roadmap	7
2 Soccer Server	9
2.1 Introduction	9
2.2 Overview	9
2.3 Cycles	10
2.4 Senses	10
2.4.1 Visual Sense Data	11
2.4.2 Audio Sense Data	11
2.4.3 Body Sense Data	12
2.5 Actions	12
2.6 Summary	12
3 Behavior-Oriented Design	13
3.1 Introduction	13
3.2 Overview	13
3.3 Development Process	13
3.4 Behaviour Modules	14
3.5 Reactive Plans	14
3.6 jyPOSH	14
3.7 Summary	14
4 Literature Survey	16
4.1 Introduction	16
4.2 Development Methodologies	16
4.3 Agent-Oriented Development Methodologies	17
4.4 Agent-Oriented Programming Languages	18
4.5 Development Tools	18
4.6 RoboCup Soccer Simulation	19

4.6.1	Self-Localisation	19
4.6.2	World Model	20
4.6.3	Communication	22
4.6.4	Synchronisation	23
4.6.5	Low-Level Skills	24
4.6.6	Positioning	25
4.6.7	Heterogeneous Players	26
4.7	Summary	27
5	Implementation	28
5.1	Introduction	28
5.2	Models	28
5.2.1	World Model	28
5.2.2	Memory Model	33
5.3	Skills	37
5.3.1	Introduction	37
5.3.2	Turn	37
5.3.3	Run to ball	38
5.3.4	Go to	46
5.3.5	Shoot	46
5.3.6	Clear	47
5.3.7	Dribble	48
5.3.8	Pass	50
5.3.9	Go To Defensive Position	52
5.3.10	Stand Up	53
5.3.11	Go To Attacking Position	53
5.4	Percepts	53
5.4.1	Introduction	53
5.4.2	Can Move	53
5.4.3	Is Dead Ball	54
5.4.4	Can Kick Ball	54
5.4.5	Can Catch Ball	54
5.4.6	Is Ball Location Known	55
5.4.7	Is In Position	55
5.4.8	Is Too Close	57
5.4.9	Is Attacking	58
5.4.10	Is Pass Available	59
5.4.11	Is In Shooting Range	60
5.4.12	Is My Nearest	60
5.5	Plans	61
5.6	Development Process	61
5.7	Initial Stage	61
5.7.1	Initial Decomposition	61
5.7.2	Iterative Design	62
5.7.3	Final Decomposition	63
5.7.4	Strategy	64

5.8	Summary	64
6	Results	65
6.1	Introduction	65
6.2	Performance Against other Available Teams	65
6.3	Performance Against Teams Using Same Behaviours	66
6.4	Summary	66
7	Discussion	68
7.1	Introduction	68
7.2	Team Performance	68
7.3	Behaviour-Oriented Design	69
7.3.1	Iterative Development	69
7.3.2	Behaviour Decomposition	70
7.3.3	Reactive Plans	70
7.3.4	Cycles	71
7.3.5	Seperation of Plans and Behaviours	71
7.3.6	Debugging	71
7.4	Summary	72
8	Future Work	73
8.1	Introduction	73
8.2	Percepts	73
8.3	SoccerServer Features	74
8.4	Debugging	74
8.5	ABODE	75
8.6	Summary	75
9	Conclusion	76
	Bibliography	77
	Appendix A Plan Files	80
	Appendix B Code Listings	84

List of Figures

5.1	Two circles centred at flags with radii equal to the distance of the flag from the player which intersect once	31
5.2	Two circles centred on flags with radii equal to the distance of the flag from the player which intersect twice	31
5.3	Player missing viewing ball while turning to look for it	39
5.4	Pass targets directly to receiver and slightly ahead of receiver . .	52
5.5	Potential area a centre forward considers to be in position using <i>home position/freedom</i> positioning approach	56
5.6	Potential area a centre forward considers to be in position using <i>starting position/home position/freedom</i> positioning approach . .	57
5.7	Potential area a left midfielder considers to be in position using <i>rectangular</i> positioning approach	57

List of Tables

4.1	Asynchronous nature of action and visual sense cycles	24
5.1	Actual world state and player's world model of angle from ball when turning past it	41
5.2	Actual world state and player's world model of angle from ball when turning at half angles - two cycle scenario	41
5.3	Actual world state and player's world model of angle from ball when turning at half angles - four cycle scenario	42
5.4	Calculations for finding optimal intercept position	44
5.5	Final Behaviour Decomposition	63

Acknowledgements

I would like to thank Dr. Joanna J. Bryson for her supervision, encouragement and support throughout the course of this project.

Chapter 1

Introduction

Over the history of artificial intelligence, there have been a number of standard problems which have been used to benchmark the current state of the art. These have included games such as chess. Such games involve decisions being made based on complete knowledge of the world state; that is that the positions and set of possible actions at any given turn are known. They are also turn based, so making the immediate effects of any action to be taken deterministic.

I tackle a problem which is emerging as a new standard problem – that of playing football. This game varies from the aforementioned traditional ones, as it does not involve complete knowledge of the world. This is because a player cannot have knowledge of everything happening in the game at any moment – they cannot tell exactly what is behind them, and their sensors involve a degree of noise. Also, the game is non-deterministic as the results of an action taken by a player can be affected by the actions of other players (for instance, if two players kick the ball simultaneously, the desired ball velocity of neither player is achieved). Also, the fact that players must be written to run independently to each other causes the emergence of the problem of getting their actions to complement each other's.

I use the Behaviour Oriented Design (BOD) development methodology for intelligent systems in my solution to the problem. This methodology places emphases on iterative development and modularity (Bryson, 2003) .

Through experience of my use of BOD, I critically analyse the methodology. This includes insight of my experience in using the methodology and identification of possible areas of future refinement for it.

1.1 Roadmap

If you would like to know about the nature of SoccerServer, read Chapter 2. For more in-depth information on the domain, I recommend you read the SoccerServer manual. If you are interested in the issues which must be addressed in implementing a team for it, and in ideas used by others in doing so, I advise

reading Section 4.6, while Chapter 5 gives some insight to my approach to these issues.

If you would like to read an overview of BOD, read Chapter 3, or for a fuller description, you may wish to read Bryson (2003).

If you are interested in the concepts used in my implementation of a SoccerServer team, you may like to read Chapter 5. If you have knowledge of BOD, and would like to get some idea of the final structure of my agents, skip ahead to Section 5.7.3.

Finally, if you would like to read my findings from my experience with BOD and SoccerServer, you may just want to read from Chapter 6 onwards.

Chapter 2

Soccer Server

2.1 Introduction

In this chapter, I describe the concept of SoccerServer and how it is used in the RoboCup tournaments. I also detail how clients written for use with the server can interact with it. This gives an insight into how my implementation of a simulated team works in terms of its interactions with the server. More information on this is given in Chapter 5, which also discusses some of the issues arising from using this set up.

2.2 Overview

The RoboCup soccer tournaments have been run since 1997 (Kitano et al., 1998). The tournaments that are run fall into two classes – those in which real robots are used, and those which are simulated. The real robot classes are generally split into tournaments using standard-platform robots, and tournaments for non-standardised robots.

In the competitions featuring non-standard robots, the main criteria for teams to be successful are related to the motor skills of the robots, such as their ability to walk or run and to kick the ball. This is because if robots are not able to effectively perform these basic actions, any more complex behaviour incorporating these basic actions will also be ineffective as a result. World perception also takes a high priority in such competitions, as even if robots effectively perform the intended actions based on their beliefs about their surroundings, if these beliefs are inaccurate, the effects of the actions performed will not be as desired. For example, if a player attempts to kick the ball with the belief that it is directly in front of them, then if it is not, the player will fail to kick it.

With standard-platform robot competitions, the issue of low-level motor skills of robots is removed as a differentiating factor between teams as these very low-level skills are already implemented in the standard platform robots which teams are provided with. Also, although world perception is still a very

important issue, the quality of raw sense data collected by robots is not a means to discriminate between teams as a result of all robots having the same sensory equipment, thus providing the same quality of raw sense data. However, individual implementations can determine *what* sense data will be received, as these implementations will determine the positions of the sensors and the directions that they are facing. The simplification of the problem leaves teams with more scope to develop higher-level strategies as a result of lower-level skills being standardised across teams, meaning that these lower-level skills are not discriminatory factors between teams. This type of tournament also places a greater emphasis on world perception because such higher-level strategies, as used in these tournaments, rely more on world perception being performed accurately.

The simulated RoboCup competitions further simplify the problem through the eradication of any issues arising from the necessity of using physical robots. This includes the need to maintain physical robots and also issues such as the fact that real robots can fall over. Two simulated competitions are currently run – 2D and 3D. The simulators are both intended to provide agents with sensory inputs which mimic those which would be received by real robots. The 2D simulator simplifies the problem further by removing the need to consider the concept of height. This means that the ball cannot travel *over* a player such that they are unable to reach it.

Despite the simplifications to the problem resulting from using a 2D simulated environment, the problem is still very complex. In fact, the problem resulting from the simplifications is more similar to that of actual *people* playing football. This is because they simply remove some of the decisions that are made more or less subconsciously by people, such as recognising that an object is a ball or adjusting balance to avoid falling over. The necessity of making all decisions which are made more consciously by football players remains as a part of the problem.

2.3 Cycles

The 2D SoccerServer, which I am using in this project, works by performing actions at set intervals known as *cycles*. Different intervals are used for different actions, such as sending players sensory information and game cycles when the world state is updated.

2.4 Senses

SoccerServer interacts with agents via UDP/IP. The server sends clients three kinds of sense data:

- Visual
- Audio
- Body

All sense data sent to players is accompanied by the game cycle to which it relates.

2.4.1 Visual Sense Data

Visual data is a representation of what a player sees, consisting of details about seen objects and their relative positions from the player. Players only receive visual sense data for objects which are in their field of view. Objects which can be viewed by players are:

- Flags - known, fixed points on the pitch
- Goals - special flags representing the centre points of the goals
- Ball
- Players

The level of detail included in visual sense data relating to the identities of objects is determined by the player's distance from the object, and the type of the object. For example, if the viewed object is another player, then depending on the distance between the players, details may, or may not include:

- The team to which the player belongs
- The player's jersey number
- The player's current velocity

There is also a degree of noise involved in received positional and velocity data. Like identity data, this level of noise is relative to the player's distance from the viewed object, with noise having a greater effect the further the player is from the object. This means that such visual sense data is not always exactly correct.

2.4.2 Audio Sense Data

Audio data is a representation of what a player hears. This can be messages from the referee, other players or coaches. Audible messages from the referee notify players of changes in the state of play, such as to announce a throw-in or free kick, and are always received by all players.

Messages sent from other players are received if a player is within a certain distance of the player sending the message. Messages sent from players can be received from players on either team. The message is accompanied by the team of the player sending the message, the distance between the message recipient and sender and, if the player receiving the message is on the same team as the player sending it, the speaker's jersey number.

Coach messages are received by all players, and are accompanied by an identifier of the team to which the coach belongs.

2.4.3 Body Sense Data

Body sense data is a representation of a player's current physical state. This information includes the number of times each type of action has been performed by the player in the game, the player's current stamina and their current velocity relative to the direction they are currently facing. This data can be used to check which intended actions were actually executed on the server by checking if the count of the number of times that type of action has been executed has been incremented since attempting to perform the action.

2.5 Actions

SoccerServer allows a number of low level actions to be executed by players. These are executed by having client software send the action in a specified format via UDP/IP to the server. The actions available are very basic, and are limited to:

- turn
- dash
- kick
- catch
- move
- turn_neck
- say
- change_view

The effects of these actions are mainly self-explanatory, with the exception of *change_view* which is used to change the shape of a player's field of vision. Only one of *turn*, *dash*, *kick*, *catch* and *move* can be performed per cycle. *turn_neck*, *change_view* and *say* can each only be performed once per cycle.

The parameters which need to be specified to perform these actions are restricted. For example, there is a maximum *kick* power and a character limit for the *say* action.

2.6 Summary

Using SoccerServer as a domain for implementing my simulated football team allows for a great deal of low-level control over players. I have given a brief introduction to means through which clients can interact with the server in order to highlight this. I will now move on to describe the methodology which I used to create agents for use with it.

Chapter 3

Behavior-Oriented Design

3.1 Introduction

In this chapter, I describe the methodology which I will use in creating my artificial intelligence football team and aim to critically analyse. I describe the development process as prescribed by the methodology and some of the key concepts that it uses.

3.2 Overview

Behaviour Oriented Design (BOD) is a “development methodology for creating complex, complete agents such as virtual-reality characters, autonomous robots, intelligent tutors or intelligent environments” (Bryson, 2003). BOD describes both the basic structure of agents created using it, and the development process which should be used when creating them.

3.3 Development Process

BOD is meant to be an iterative process. This lends itself well to rapid prototyping, and allows developers to decide on what needs to be implemented or improved in an agent based on the most recent development iteration of it.

Iterations are to be carried out by carrying out an initial decomposition of the intended agent, identifying an area to implement, and then implementing that area. This is to be repeated, revising the specification after each iteration (Bryson, 2003).

When using BOD, code should be self-documenting, thus maximising its maintainability and readability (Bryson, 2003).

3.4 Behaviour Modules

According to the principles of BOD, agents have a set of *behaviour modules* which an agent uses in order for them to act as they are specified to. These behaviour modules can incorporate senses, basic actions, more complex decision-making and some sort of state (Bryson, 2003).

Behaviour modules can be implemented in any language, but due to their nature, object oriented languages lend themselves well to this purpose as BOD builds upon object oriented principles.

3.5 Reactive Plans

As well as behaviour modules which are implementations of what an agent can do, agents also have *reactive plans* which specify when they should do what. Reactive plans use a combination of senses and actions implemented in behaviour modules in order to decide what the agent should do (Bryson, 2003). These sense and action primitives can be built upon to form more complex plan elements which can be used in plans. These are:

- Action Patterns
- Competences
- Drive Collections

Action patterns are a list of actions to be executed in a specified order. Competences are like action patterns, but each action in a competence may optionally have triggers for each action in them and a goal. Triggers are a set of criteria which must be satisfied in order for an action to be executed and goals define what should be achieved in order for the competence to be interpreted as being successful. Finally, drive collections are the top-level plan elements. Drive collections are similar to competences, but they have no goals, as they never terminate and are used in every cycle.

These reactive plans are written as POSH scripts which have their own syntax and are independent of any programming language.

3.6 jyPOSH

jyPOSH is a system written for the implementation of POSH reactive plans. It is written in jython, which provides good integrability with the Java programming language.

3.7 Summary

BOD uses an iterative design process and strongly promotes decomposing agents into their individual behaviours. It allows for reactive plans to be used which

specify which actions and senses supported by these behaviours should be used in different situations.

In the next chapter, I describe my findings from literature which I read to help with the implementation of my football team. This includes information on design methodologies other than BOD which are also candidates for use in such a project.

Chapter 4

Literature Survey

4.1 Introduction

In this chapter, I describe my findings from a variety of literature. This includes that on various development methodologies, which allows for comparisons with BOD to be made.

The chapter also includes the reported findings of, and strategies used by other people who have worked with SoccerServer. This helps shape the implementation of my own football team through taking the approaches used by others into account. I also detail some of the issues that have been identified as being important when implementing a SoccerServer team.

4.2 Development Methodologies

There exist many software development methodologies. This is partly because different methodologies cater for different kinds of projects. One reason for this is that different methodologies apply different overheads to a project, and demand different levels of procedural rigour which are not necessary in all projects. Another reason for the existence of so many development methodologies is that no methodology is perfect, so there is always *room for improvement*, and so new methodologies are created in order to address this.

There also exist a number of development methodologies, including BOD, specifically for use with agent-based systems. For this reason, it makes sense to research alternative agent-oriented development methodologies. This is because through a knowledge of these other methodologies, parallels can be drawn with BOD through which the methodologies can be compared and contrasted.

When choosing an appropriate development methodology for a general software development project, there are a number of factors to be taken into account. Cockburn (2000) says that the main two are the number of people working on the project and the criticality of the system. In the particular case of this project, there will be one person working on it, with a low criticality, as it

is being conducted for research purposes. In contrast, Yusof et al. (2011) outline uncertainty and complexity, system quality and the scope of the methodology as the main criteria to be taken into account when selecting a methodology. Both proposed methods for selecting a methodology break down these main criteria into further sub-criteria.

4.3 Agent-Oriented Development Methodologies

There exist a number of development methodologies specifically design for agent-oriented systems. Padgham and Winikoff (2009), Bresciani et al. (2003), DeLoach et al. (2001), Wooldridge et al. (1999) and Bryson (2003) describe a few examples. However, the emergence of agent-oriented development methodologies is relatively recent, and so these methodologies are not mature.

Abdelaziz Tawfig (2008) suggests that agent-oriented development methodologies fall into three main categories. These are purely agent-based methodologies, methodologies which build on object-oriented (OO) methodologies and knowledge engineering-based methodologies. Wooldridge et al. (1999) argue that OO methodologies are unsuitable for the development of agent-oriented systems. They argue that the only effective development methodologies for agent-oriented systems are those created specifically for that purpose. DeLoach et al. (2001), however, take a different viewpoint as their MaSE methodology builds upon existing OO principles.

Abdelaziz Tawfig (2008) highlights how currently-existing agent-oriented methodologies have been designed with agent-based systems with particular characteristics in mind. This raises the question of how the characteristics of a particular agent-oriented system can be measured in order to decide upon a development methodology to use for it. Luck et al. (2005) recognise this, and also recognise that traditional software engineering metrics for software projects are incompatible with agent-based systems; they highlight the necessity of establishing a set of metrics which can be used for such systems. However, Abdelaziz Tawfig (2008) points out that there is no general agreement between methodologies on how to characterise agents. This is problematic when selecting a methodology for a particular project, as there is no one set of criteria, applicable to every methodology, that can be used to assist in making the decision.

In order to select a methodology specific to agent-oriented software, it is useful to identify features specific to the proposed methodology. Dam (2003) proposes that agent-oriented development methodologies can be compared on concepts, notation, process and pragmatics. Of these, pragmatics appears to be the least relevant to this particular project, as it is more of a business-oriented criterion. However, issues such as the difficulty to familiarise oneself with the methodology remain important.

One defining feature for an agent-oriented development methodology is its *scope*, meaning the stages of development covered by the methodology. Luck et al. (2005) express the belief that development methodologies should encompass all phases of the software-engineering lifecycle. By this, they mean phases

from requirements analysis, through to post-deployment analysis. Abdelaziz Tawfig (2008) identifies that although some do, many agent-oriented methodologies do not incorporate an implementation phase. A likely reason for this is the lack of accepted agent-oriented programming languages which can be specified for the implementation phase.

4.4 Agent-Oriented Programming Languages

Luck et al. (2005) suggest that agent-oriented methodologies should be implemented using a bespoke agent-oriented programming language. They argue that using an imperative language, such as would be the case when using a development methodology which builds upon an OO methodology, is inappropriate for such systems. They do however, identify that it is desirable for some concepts from OO design to be included in an agent-oriented language. Luck et al. (2006) point out that agent-oriented programming languages have had very limited use outside of research. The argument for the use of agent-oriented programming languages largely centres around the possibility of using agent-related concepts as constructs in the language rather than having to artificially implement them in non-intuitive ways. This would enable implementations of agent-oriented systems to be much more closely aligned to their designs.

Part of the argument over whether to devise an entirely agent-oriented programming language for use with an agent-oriented methodology is related to the usability of legacy code. As there are many mature OO languages, there exist libraries of code which would prove useful in agent-oriented software. The use of a language which extends such an OO language would mean that legacy code written in it could be easily used in an agent-oriented system. Luck et al. (2006) proposes that for a new agent-oriented language to be successful, it should be able to interface with legacy code written in other languages. This would give the new language the advantages related to being able to use agent-oriented constructs, while keeping legacy libraries available.

4.5 Development Tools

For a development methodology to be successful, it is useful for it to have appropriate tools which support it.

Garcia et al. (2011) highlight the relationship between agent-oriented methodologies and the tools that support them. They describe the spectrum from how some “methodologies” are just a set of tools, to others which are a set of guides for how to follow the methodology, with no supporting tools. They also argue that without such tools, the functionality of a methodology is severely compromised. This view of tools for use with a methodology as being an integral *part* of the methodology itself is contentious. This is because, although in some cases, development tools are created in parallel to formulating the methodology itself, it appears far more common for development tools to be produced *after*

the methodology has been put to some use. This is because it is not until the methodology is trialled, that the nature of the tools which would complement it become apparent. The question of whether development tools are a constituent part of a development methodology is also dependent upon how one defines “development methodology”.

Although the tools for use with a particular methodology are not necessarily a constituent part of the methodology, these tools, in the majority of cases, improve the usability of the methodology. They also can increase the efficiency with which development can be conducted using the methodology. This can be as a result of the tools assisting with documentation or of them assisting with the production of code.

The effectiveness of tools for use with development methodologies is partly dependent upon the phases of the methodology in which the tool can be utilised. DeLoach et al. (2001) describe the agentTool tool which has been developed for use with the MaSE methodology and can be used during analysis, design and, to a limited degree, implementation phases of the methodology. Padgham (2005) describes tools which help enforce consistency over dynamic, iterative development in the Prometheus methodology, but do not have any application in the implementation phase. This is relevant to BOD, in the respect that it is also an iterative methodology. However, the tools described only extend to support specification and design of a project. The Advanced Behavior Oriented Design Environment (ABODE) tool is meant to not only allow for plans to be visualised, but also generate code which implements these plans.

4.6 RoboCup Soccer Simulation

Many approaches have been taken in implementing RoboCup Soccer Simulation (RCSS) teams. Of these implementations, differing emphases are used on different areas of a team’s behaviour. These range from low-level focuses, such as the task of players determining where they are on the pitch (Penedo et al., 2003) to advanced strategies involving players exchanging positions during the course of a match (Reis and Lau, 2001). In order to produce a competitive RCSS team, it is useful to research the areas in which previous research has been conducted by those creating their own teams. Also, some teams, such as CMUnited99, have made their source code available. It may be useful to use some of this source code in my own team, as others have done, in order to avoid having to implement some of the lower-level intelligence, such as self-localisation. However, it is important to avoid using code which will compromise the experience of using BOD.

4.6.1 Self-Localisation

The problem of players being able to estimate their position on the pitch, otherwise known as the self-localisation problem, exists in RCSS. This is because players are not given information on their absolute position. They do, how-

ever receive information on relative positions from any stationary flags or non-stationary objects within their field of vision. The positions of the flags are known. Clearly, given this information, it makes sense for players to self-localise using the relative positions of flags to themselves; however, there are a number of methods of doing this.

Penedo et al. (2003) describe a method whereby the pitch is divided into a number of squares. Then for each square on the field, a probability is calculated that the player is in that square, given the available information on relative positions of stationary objects. The solution appears to give reasonably accurate results, however, it is highlighted that, due to having to make calculations based on many possible locations of the player, the method is relatively computationally expensive.

Amiri et al. (2004) describe a method where the intersections of polygons derived from all viewed flags are used to calculate the player’s position. They claim that this method is relatively computationally undemanding, and yields greater accuracy over “many” other teams’ methods.

Polani et al. (1998) touch briefly upon the issue of self-localisation, simply by stating that the agent’s location is calculated “in a straightforward way from the flags and lines seen”. Similarly, Stone et al. (1999) state that “The agent’s position can be determined accurately by using the relative coordinates of one seen line and the closest stationary object”. Although, these do not describe precisely the methods used, from the on-pitch features mentioned, it is likely that some sort of triangulation method is used, such as that detailed in Bowling et al. (1996). Since this method is performed purely using geometric calculations, it is very reliable, provided at least two flags are seen at any one time. However, it is noted that the calculations are computationally expensive, which must be taken into account, given the time-sensitive nature of decisions made by players.

Although the descriptions of these methods highlight their benefits and weaknesses to some degree, no comparison of the methods could be found. Such comparisons do exist for the domain of self-localisation of *real* robots (Kose et al., 2008). It is notable that the methods detailed in these comparisons are mainly probability-based. Although these comparisons may be useful to some degree, they are not applicable to the domain of RCSS. This is because of the absolute knowledge that a viewed flag is a particular flag (at a particular location), which cannot be assumed with sensor information collected from real robots.

4.6.2 World Model

As well as players knowing their own position on the pitch, it is also important for them to be aware of the locations of other game objects, such as the ball and other players during the course of a game. In fact, it is possible for players to *not* know their own position on the pitch, but to be aware of where other game objects are relative to them, as this is the format in which they receive visual information. However, a player’s knowledge of their location is useful when considering tactics, and even when making simple decisions such as avoiding

running off the pitch.

Players only receive visual information on objects within their field of vision. This means that in any one set of visual information received, information on the majority of game world objects is not included. It is therefore sensible to construct some sort of *world model* in which information, such as positions of game world objects, is stored so that players have some knowledge of game objects which they not looking at.

There are two obvious approaches for storing world models. The first is to store the positions of game objects *relative* to the player. Using such a system would be very simple to implement as visual information is received by players is relative to their position. However, when a player moved, it would be necessary to update the locations of all other world items to take into account this movement in order for the model to maintain its accuracy. The other option for storing world models is to store *absolute* positions and other information of game objects in the game world. Although using this approach would involve a little more work in translating relative positions into absolute ones, given the knowledge of the player's position, this is trivial. This approach makes updating the world model upon the player moving less computationally expensive. This is probably the reason for almost all RCSS teams electing to use world models which are structured in this manner.

One issue which affects the calculation of world models is the existence of *noise* in visual information sent by the RCSS. This noise is intended to simulate the noise which is received from visual sensors on real robots. As such, more noise is perceived on objects which are further away than those which are closer to the player. This noise not only affects the perceived locations of viewed objects, but also other information such as their direction, velocity and information such as the identity of a particular viewed player, and even the team for which they play.

In order to take this noise into account, Stone et al. (1999) describe methods for inferring the information which is not received. For example, if a player is viewed in one set of received visual information, and then is not viewed in the following set, but a player with missing information is viewed in a position similar to where the original player was predicted to be, it is assumed that this player is the originally viewed player. Another example is when a player is viewed, but is too far away to perceive their velocity. In this case, the difference in their position between two sets of visual information is used to make an estimate of their velocity. These methods provide players with means to improve their world models from just consisting of the visual information they receive by applying some context to the information.

Another issue affecting the world model is the behaviour of objects which have been viewed, but then cease to be within the player's field of vision. Bowling et al. (1996) describe *predictive memory*, where the current positions of the objects are estimated as a result of information collected when they were last seen. A *confidence* value is assigned to each object with predicted values. This confidence value represents a player's confidence that the information on an object in their world model is correct. This confidence deteriorates over time in

which the object is not seen, because over time, the discrepancy between predicted information and the actual world state tends to increase. It is reported that, in the domain of RCSS, using this *predictive memory* is advantageous over only acting on what is directly observed. Full advantage can be taken from using such a method only if players take into account the *confidences* associated with objects in their world models when making decisions; otherwise, calculating these confidence values would be of no value.

de Boer et al. (2001) describe a similar method to Bowling et al.’s *predictive memory*, where positions of previously viewed objects are predicted based on their observed velocities, and confidence values are calculated for these positions. They claim that using this method results in a player being aware of the approximate positions of 17 of the 22 players on the pitch at any one time. This level of knowledge of other player’s positions on the pitch clearly allows for much better informed decisions to be made.

There is obviously some variation in how world models are used and calculated in different teams. However, a system using absolute positioning appears to be the favoured structure, probably due to the ease of manipulation and its intuitiveness. Also, making some sort of predictions on information about objects which are not in immediate view seems to have an overall positive impact on the effectiveness of a player’s world model.

4.6.3 Communication

Visual information is not the only way that players are able to gather information about their surroundings. Players are also able to *say* things to each other. Like vision, the aural model does not give players full knowledge of all messages. That is, only messages sent from players within a certain range can be received, and the only information accompanying the messages themselves are the direction from which the message came relative to the player. Also, under the current default configuration of SoccerServer, messages are limited to ten characters in length. There is also only one channel of communication, meaning that only one message can be sent from one of all the players on the pitch at any one time. Although these *say* and *hear* abilities do give players an additional source of information on their surroundings, the restrictions involved make it unfeasible for players to utilise them in order to share their world models with each other in their entirety (Noda et al., 1997). The ability for players to communicate with each other can still, however, be useful.

Igarashi (1999) describes a strategy for communication within a team where only two players communicate with the rest of the team. In this system, the goalkeeper communicates with the defence in order to organise where they position themselves, and similarly, a central midfielder communicates with the forwards to organise where they position themselves. This technique of only using a couple of players to communicate helps to avoid players having to deal with receiving possibly conflicting information or commands from teammates.

Stone et al. (1999) use communication for more varied and complex reasons. One of these is informing teammates of parts of a player’s world model.

This enables players to have a more accurate world model with regards to objects which they cannot see, or are far away from by incorporating the extra information received from teammates into this world model. They also use communication to coordinate the teams defence by organising which players should mark which opposing players. Communication is also used to organise players in in set plays. In order to achieve this level of coordination, they implemented a protocol, as described in Stone and Veloso (1999), where messages include the players for whom they are intended, the identity of the sender, timestamps, and information on strategies which the player is currently following. This protocol allows for players to effectively determine whether a message is applicable in their particular situation, and also provides a way to prevent against opposing players sending messages to deliberately mislead them.

de Boer et al. (2001) describe how they use the ability for players to communicate, by having players with a good view of the world tell other agents important aspects of what they can see. This allows players to incorporate this extra information on their surroundings into their world models. This means that their world models can be more accurate than if they were calculated purely from the visual sense data received by themselves.

The variety of uses for which communication has been used in RCSS teams underlines its versatility and power. However, due to the bandwidth restrictions on messages, it is important to carefully select how best to use communication within a team.

4.6.4 Synchronisation

In RCSS, players can perform actions once per 100ms *cycle*. Players are sent visual information once per 150ms.

This means that the visual information received most recently by an agent will not always be correct at the time of selecting an action. It also means that if the server *receives* multiple commands from an agent within the same cycle, only the first sent command will be executed as a result of this *clash*. For this reason, many teams employ some form of strategy to account for this.

de Boer et al. (2001) describe how their agents take into account when messages are received from the server, and use these time intervals to decide when messages should be sent to the server. They do this rather than sending messages at exact intervals because it takes into account the period of time taken to send messages to the server, and also accounts for if the server is running fast or slowly. This synchronisation strategy also means that each selected action can be based on the most recently received information from the server. After messages have been sent to the server, their agents check that the action has been executed by analysing information received from the server before sending more messages, so avoiding *clashes*.

Players as described in Stone and Veloso (1998) and Amiri et al. (2004) update their world models to a predicted new state after actions have been performed. This means that when selecting actions between having performed an action and new visual information being received, the probable actual world

Table 4.1: Asynchronous nature of action and visual sense cycles

Time (ms)		
0	Visual Sense	Action
50		
100		Action
150	Visual Sense	
200		Action
250		
300	Visual Sense	Action
350		
400		Action
450	Visual Sense	
500		Action
550		
600	Visual Sense	Action
650		
700		Action

state can be taken into account. A problem with this method occurs if clashes occur between actions sent to the server. In such a situation, the player would update their world model based on actions which weren't actually performed, thus rendering it inaccurate. Selecting actions based on an inaccurate world state may lead to the player performing undesirable actions, such as kicking the ball in the wrong direction.

Polani et al. (1998) describe a more advanced approach. As with the aforementioned strategies, players update their world models between visual information being received based on the actions that they perform. In addition to this, they perform comparisons between the perceived world states and their predicted world states. This enables them to measure the accuracy of their predictions, which can vary as a result of a number of factors affecting when the server receives commands from the player. Players are then able to make decisions using this calculated accuracy of predicted world states. For instance, an action for which the accuracy of the believed world state is very important can be set to only be selected when confidence in the world model is high. This helps agents to avoid making very costly mistakes as a result of having an incorrect world model, while still allowing for decisions requiring a good world model to be taken in the right circumstances.

4.6.5 Low-Level Skills

As described in Section 2.5, the RCSS only accepts very low-level action commands – *kick*, *dash*, *turn*, *turn_neck*, *move*, *say*, *catch* (for the goalkeeper), and *change_view*.

Although these commands are enough to implement players, their low-level

nature requires higher-level actions to be implemented on top of them. For example, there are no *pass-to*, *shoot* or *dribble* actions. This gives a greater level of control over players, as these actions can be implemented as the programmer sees fit. A side-effect of this low-level control is that poor implementation of low-level skills can potentially prove much more detrimental to a teams performance than poor high-level strategy. This is because, even if a team has very advanced strategic plans, if they cannot dribble, pass, shoot or perform other basic skills effectively, they will not realise the potential of these strategies. This theory is reinforced by the findings detailed in Stone et al. (1999), saying that their team’s strategic abilities only really became apparent when faced with opponents with a similar level of low-level skill.

It is important to note the exact effect of the low-level actions which are sent to the server, and possibly to implement skills which have effects that are somewhat more intuitive at a relatively low-level in the player implementation. For example, the *kick* command alters the ball’s motion in the direction of given angle, and to a magnitude determined by the kick power. The command does *not* cause the ball to be kicked to result in a given speed in a given direction. Yang et al. (2002) implements a low-level kick skill in which the ball’s current velocity is taken into account. This results in a skill in which the desired ball-velocity can be specified, and then a kick can be performed to result in the given ball velocity. This simplifies the implementation of higher-level skills involving *kick*, as they do not have to take into account the server’s interpretation of the action.

The nature of the commands which can be sent to the server leaves the decision open as to how advanced these skills are. For example, *dribble* could be implemented such that it dribbles in a given direction or the skill could be made to incorporate avoiding opposing players. These sort of decisions lead to many people making the design choice for players being implemented using a multi-layered architecture, whereby “higher level” skills are layered on top of “lower level” ones. A good example of such a design is detailed in Igarashi (1999). Here, *actions* are divided into 4 clear *levels*. This allows them to reuse some lower-level skills in higher level actions, building up to very high-level actions such as *offense* or *defense*. This means that high-level strategies do not have to be implemented in terms of very low-level actions.

4.6.6 Positioning

As in real football, it makes sense in RCSS games for teams to play with a particular *formation*. This involves different players playing primarily in different areas of the pitch. For example, a defender will tend to position themselves closer to their own goal than a centre forward. This helps to conserve players’ energy, as it means that players have to cover less of the pitch. It also means that players are more likely to be in space to receive passes, and be distributed around the pitch such that the opposing team will not have large areas of the pitch to attack in unopposed.

A number of decisions must be made when deciding on a team’s formation

and how players should act within a formation. One such decision is to decide upon the area in which a player operates. Wang et al. (2008) describe a strategy in which the pitch is divided into rectangles, and a player in a given position operates in a certain subset of these rectangles when their team is defending, and possibly a different subset of the rectangles when their team is attacking. This strategy gives players scope to overlap in their positions in order to mount complex attacking moves. It also means that players from different positions can assist with attacking and defending by positioning themselves where they can be of use, while avoiding players all converging to the same places on the pitch.

Stone and Veloso (1998) use a similar strategy in which players are assigned a *home range*, which is the area of the pitch which a player should operate around. The players then, based on game circumstances, select a position from their *home range* to set as their *home coordinates*, which is where they should go. Players are also assigned an area of the pitch called their *maximum range*, and are considered to be *out of position* if they stray out of this. Like with Wang et al.'s approach, this affords players flexibility of positioning, while still ensuring that they stay *in position*.

Both of these approaches incorporate the ability for teams to change their formations dynamically during the course of a match. This enables them to alter their strategy in order to exploit weaknesses identified in their opponents, or to combat threats caused by the opposition's strategy. It also allows them to change their strategy based on the match situation. For example, a team may choose to change to a more attacking formation when losing towards the end of the game. Of course, for a team to alter its formation, players affected by the change need to be aware of the changes so that cases do not arise in which different players attempt to play based on the team using different formations.

The determining of where a player should position themselves more precisely than an area of the pitch based on their role in the team must also be considered. In order to do this, the positions of the ball, teammates and opposing players may be considered. This is a complex problem in which many decisions must be made. For example, should a player try to keep in position when they have the ball? and should players avoid being too close to teammates?

An example of a team's strategy for such positional play is described by Nakagawa et al. (2000). They describe their defenders' behaviour when the opposition are attacking. They organise themselves in a line in order to increase the likelihood of an opposition player being caught offside, and along this defensive line, players position themselves to block opponents' routes to goal. Tactical positioning, such as this can be bases for higher-level strategies employed by teams, as seen in real football.

4.6.7 Heterogeneous Players

It sometimes makes sense for players with different roles to exhibit different behaviours. In the most obvious case, the goalkeeper should clearly behave differently outfield players in that they should attempt to catch the ball when

within their penalty area, and should not make runs up field. Also, players in RCSS are heterogeneous in that they have different physical attributes.

Reis and Lau (2001) implement their team with players being assigned different player *types*. These different player types have differing tendencies to exhibit different behaviours. This means, for example, that defenders can have less tendency to perform risky actions than forwards. Using these player types, they are able to get players to behave differently from each other in a manner in which their behaviours complement each others’.

Gabel and Veloso (2001) describe in-depth an advanced machine-learning based method for selecting which players with which physical attributes to play in each position. This means that players can be selected to play with particular roles for which their physical attributes are suited. Also, due to the machine learning-based approach used, these decisions can be based on actual perceived effects of past player role selections. This can lead to selections being made in order to achieve benefits which could not be recognised by intuitive theory.

4.7 Summary

In this review of literature, I have explored development methodologies for agent-oriented systems. This reinforces the importance not only of using a good methodology, but using the *right* methodology for the task in hand. This means that my analysis of BOD needs to be in terms of it as a methodology specifically for the development of an artificial intelligence football team. However, I am able to extrapolate this experience to a degree in order to analyse it in a more general sense. The research conducted also highlights the importance of useful development tools. This helps to outline the importance of usability and power of the ABODE tool.

In researching strategies and issues relating to RCSS teams, I found a very wide range of approaches and ideas. This made me more aware of the vast number of problems to be addressed, and potential strategies posed by the task of creating an RCSS team. The variety of approaches used caused me to reassess some decisions to be made during implementation by presenting me with some ideas which I hadn’t previously considered. The research carried out on RCSS teams also showed how highly-complex team behaviour can be achieved as a result of a combination of many simple behaviours being combined. This lends itself very well to the principles of BOD.

In the next chapter, I describe my implementation of my simulated football team, which was influenced by the knowledge obtained from this review of literature.

Chapter 5

Implementation

5.1 Introduction

In this chapter, I describe in detail how my team was implemented. As well as information on concepts which actually were implemented, I also include some discussion on how decisions on which concepts to use and how to implement them were reached. In addition to information on the concepts which were implemented, I also give an overview of the development process which was followed in Section 5.6

5.2 Models

5.2.1 World Model

In order for a player to successfully interact with their environment, it is essential for them to have some sort of understanding of the current state of this environment. For example, a player can only effectively shoot if they are aware of the location of the goal. This understanding of a player's environment takes the form of a *world model* – an agent's internal representation of their beliefs about the state of their environment.

As discussed earlier in Section 4.6.2, there are two obvious approaches for possible world models - *relative*, where all information stored is relative to the player's position/velocity, and *absolute* where all information stored is relative to a fixed world coordinate system. At first, I implemented a very simple *relative* model, going against the more popular *absolute* option. I opted for this approach, as it is more aligned to the structure of the sense data received by a client, and so it would be quicker to implement. This was so I could start to implement some very basic behaviours, and start to use jyPOSH to formulate plans very early on in development. I was always mindful that it was most likely that, unless unexpectedly good results could be seen to be achievable using a *relative* model, I would switch to using an *absolute* world model later in

development.

This relative world model which I initially implemented was very basic. It worked by simply parsing received sense data, and constructing internal representations of the world objects represented by this data. These world model objects contained only the details identifying the object received in the sense data, which could be complete or incomplete, the distance of the object from the player, and the direction it was seen at relative to the player.

When attempting to implement *very* basic skills, such as running towards a ball that the player was facing, the relative model was moderately successful. However, the *synchronisation* issue present in the SoccerServer, whereby actions are executed asynchronously to sense information being received became problematic. An example of this is outlined here:

- Player receives visual sense data
- Player uses received sense data to add a viewed goal to its world model, using the goal's distance and angle from the player
- Player turns 45°
- Player's world model now has the goal's angle from the player wrong by 45° as no new visual sense data has been received since turning
- Player attempts to shoot, but aims 45° away from the goal as a result of having a misleading world model

The above example applies to many similar cases when sense data is received, then the player's world model is invalidated by an action, and another action is executed before more sense data is received. The problem could be overcome by updating the world model upon performing an action to compensate for the expected effects of that action. For example, this would mean when the player executes a *dash*, the distances and directions of all objects in the world model would be updated to an estimate of their relative distance and directions from the player after the movement of the player resulting from the *dash*. However, there are problems with this solution. Firstly, the exact results of a player performing a particular action cannot be reliably predicted. For example, this can be as a result of the player bumping into another player or of the noise applied to movements of all objects in SoccerServer. This means that the estimates of the new relative positions will be inaccurate. Another problem with the approach is that, as there are many objects in the game world (53 flags, 2 goals, 22 players and 1 ball), there is a large quantity of world model data to recalculate on every cycle. This adds a large computational overhead to each game cycle, reducing the efficiency of agents, which must be kept reasonably high in order for action selection to be carried out in time for each game cycle.

Another issue with using such a world model is a result of the fact that players could only be made to make decisions based on their situation relative to other world objects. Although this meant that they could infer the positions of these objects relative to the pitch as a whole based on viewed flags, these

positions would have to be calculated every time such information was required. As it is important in football for players to have an idea of their position on the pitch at most times during the game, such as to maintain a formation and to not run off the pitch or offside, the frequency for this calculation being necessary is very high. Again, this is an issue because of the computational intensity of agents making fundamental inferences about their environment every cycle.

As a result of the problems involved with using a *relative* world model, I resolved, as expected, to switch to using a *absolute* model. As well as solving the aforementioned problems, I also believe this approach to be more intuitive to work with. This is because it is more closely aligned to the way the game monitor I was using displays games, with the pitch being viewed from an aerial perspective, rather than a first-person perspective of a player. Also, I found it more intuitive to formulate strategies for players by using absolute pitch locations.

At first, I retained the relative world model to be used alongside an absolute world model. The reasoning behind this was that this relative world model could be used in certain situations where only relative information was necessary. This can be an advantageous strategy in circumstances in which a player's own position has been calculated inaccurately. This is because in such scenarios, the player will also have calculated the positions of all other objects inaccurately due to these calculations being based on the results of self-localisation. This means that decisions could be made relating to the true state of the world, even if the absolute world model was inaccurate.

I implemented the absolute world model by having a player first calculate their own position on the field, and then extrapolating the absolute positions of other sensed objects from this position. This, of course, required a method for players to calculate their own position. A number of strategies for self-localisation were discussed in Section 4.6.1, however a number of these which used methods based around calculating probabilities of being in different locations such as those proposed by Penedo et al. (2003) and Amiri et al. (2004) appear to be attempting to solve the problem where the reliability of sense data is not as great as that provided by SoccerServer. This is because in SoccerServer, the identity, and thus position, of a viewed flag is guaranteed to be correct, whereas this cannot be guaranteed from a sense processed from a camera input on a physical robot. This is the reason why a *probability* based approach may be used – to compensate for any such errors that may occur. Because of this, and for the sake of simplicity, like Polani et al. (1998), Bowling et al. (1996) and Stone et al. (1999), I decided to take an approach which used purely geometric calculations.

Initially, I used a method which relied on a player viewing two on-field flags. This meant that sense data would include the player's distance from each of these flags. If more than two flags were viewed, the closest two would be selected to be used in positional calculations. This was because according to the visual model in SoccerServer, the closer objects are, the less effect noise would have on sense data about them, and so sense data on the closest two flags is the most reliable. As the positions of on-field flags are fixed, the absolute locations

of these two viewed flags are known. This meant that it was known that the player's position lay on a circle centred at one of the viewed flags with a radius equal to the player's distance from this viewed flag. As this was known about two flags, it was then also known that the player's location must be on one of the intersections between the circles around these two flags. This either meant that there were one or two possible locations at which the player could be situated. If there was only one location at which the circles intersected, then this was taken to be the player's location. If there were two intersections, the closest one to the player's last known location was selected. This follows the assumption that a player will not move large distances between two sense cycles, which is reasonable, provided the player does not use the *move* action, where they instantaneously move to another location on the pitch.

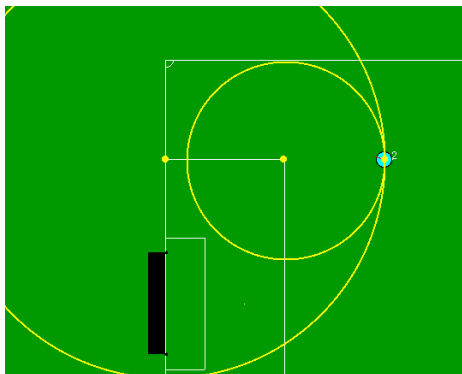


Figure 5.1: Two circles centred at flags with radii equal to the distance of the flag from the player which intersect once

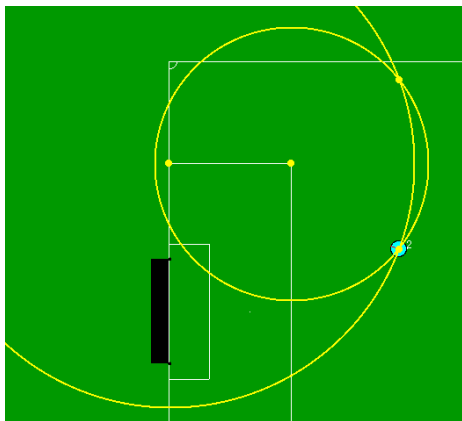


Figure 5.2: Two circles centred on flags with radii equal to the distance of the flag from the player which intersect twice

It did occasionally turn out that this method yielded false results when the circles intersected at two points very close to each other, and the player had moved to the intersection which was not the one closest to the player's original position. This meant that the incorrect point of intersection of the circles was selected to be used as the player's location. When this miscalculation was performed once, the discrepancy between the player's believed location and actual location was small, but still enough to select an inappropriate action, such as attempting to kick the ball when they were not close enough to do so. However, when one such miscalculation took place, the incorrect belief of the player's location could cause future selections of intersection points of circles based on this believed location to magnify the error. This could mean that a player's believed location could become wildly removed from their actual location, and cause them to wander away from where they are trying to get to on the pitch.

I addressed this problem by adjusting the method for calculating players' positions to use the sense data collected on *three* flags. This meant that, although it was still possible that these three circles could intersect at two points, the likelihood of this was hugely decreased. Because of this, even in the unlikely event that a player's location is calculated incorrectly, it is very unlikely that an incorrect position will be calculated on the next cycle because of the very low probability of two points of intersection being calculated. A drawback of this method is that an estimate of a player's position can only be calculated when visual sense data contains information on at least three flags. However, the only times that this is normally not the case is when a player is either off, or very close to the edge of the pitch, and facing away from it, which is not a position which a player should be in very often in a game. If such a situation does occur though, some mechanism must be used to ensure the player moves himself to view enough flags to reassess their position on the pitch.

In order to calculate the positions of viewed objects, it is necessary to not only know the player's current *location*, but also to know the *direction* in which they are currently facing. This is because even if a player knows that object x is a distance d away from them and at an angle of α , if only the player's *location* is known, x could lie at any point on a circle of radius d centred at the player's location.

The method I used to calculate the direction that a player is facing works as follows:

- Select the most distant viewed flag
- Calculate the angle at which this flag is from the player's believed location
- Subtract from this angle the relative direction of the flag from the player, as contained in sense data

Here, the most *distant* viewed flag from the player is selected to be used in the calculation, rather than the *closest* viewed flag, as used when calculating the player's position. This is because, although the effect of noise on the positions

of viewed objects increases as they become more distant, inaccuracies in calculations of players' positions have less of an effect on the estimated direction of more distant flags from the player than on those which are closer. However, as this method relies on an estimate of the player's position, if this estimate is inaccurate, the estimate of the player's direction will also be inaccurate.

In order for the team to be most successful, I decided that the best approach was to use a world model which was as complete as possible. This uses the assertion that the best decisions can be made given the most complete information because as many factors as possible can be taken into account. However, the use of complete information is impossible given the setup of SoccerServer. Because of this, and in order to make as much information available as possible when making decisions, I decided that the locations and velocities of all viewed players and the ball should be included in a player's world model.

5.2.2 Memory Model

As well as having an effective world model, it is also important for players to exhibit some form of *memory* to augment this world model. This is because if a player's world model includes only information contained in the most recent sense cycle, they will never be able to make decisions based on objects which were not sensed in that cycle. Using memory means that objects sensed in previous sense cycles can persist in a player's world model.

In initial iterations of my implementation, players had no memory model. That is, upon receiving a new set of visual sense data, the information collected in previous sense cycles would be disregarded, meaning that the only world information that could be used to make decisions was that which was received in the most recent sense cycle. This worked well in the sense that the information that players acted upon was always the most recent available information. However, it soon became apparent that it would be beneficial for players to be able to draw upon information sensed in previous sense cycles to help them in their decision making as this would mean that they would not always need to be facing an object in order to have some awareness of it.

A very simplistic approach to developing a memory model would be to add any sensed object to a player's world model and keep it there. This would mean that sense information from every sense cycle in the course of a game is included in the player's world model, which would mean that players would be aware of objects that they had viewed in previous cycles.

This simplistic approach is flawed in that if an object is added to a player's world model every time it is viewed, the world model could come to contain many duplicates of the same objects, meaning that it would be unclear as to which instances of these objects to take into account when making decisions. This problem could be solved to a certain degree by replacing any object in a world model with one based on more recently sensed information on the same object when it is received. This would work without issue for sensed objects with complete identity information, as it would be clear which object to replace in the world model. However, it would not be so straightforward for sensed objects

which could not be identified exactly, such as a player whose jersey number is unknown. If this were the case, it would be unclear as to which object in the world model to replace, or if it was an as-yet unseen object which should be added to the model.

A possible solution for this problem would be to attempt to infer the identities of players for which incomplete identity information is received, like as described in Stone et al. (1999). This could be done, for example, through assuming that any viewed player with unknown identity information is the player in the world model closest to the position of the viewed player. This can be an unreliable strategy because players which are close to each other can be confused with each other, as can players which have moved significantly since they were last viewed. Such confusion can be particularly detrimental to a team's performance if a player who is significantly far away enough for visual sense information to not include their team, is confused with a player from the opposite team. This is because a player's decisions can be drastically different for if a player in their world model is on their team, or on the opposition's. For example, this may lead to a player attempting a pass to an opposition player under the belief that they are a teammate.

An alternative solution is to not use sense data with incomplete identity information to update the world model at all. This would mean that the world model would never store information for one player which should actually relate to another player, possibly for the opposite team. However, it also means that some sense data is disregarded, and so the world model used would not take full advantage of all received sensory information. I chose to use this approach because, after trialling the other ideas, it proved that having a lesser volume of more accurate information on players in the world model was more advantageous than having more information, some of which being misleading. There are many examples of this being the case, such as situations in which players pass to opposition players, believing that they are on the same team, and players straying offside, with the belief that a teammate who is offside is an opposing defender. However, cases in which this lack of uncertain information is a disadvantage must also be noted. These include scenarios in which a player does not make a pass to an open distant teammate as a result of a lack of identity information on the distant player causing them to not be added to the world model.

A memory model in which all objects are retained indefinitely until they are updated can be useful as it maximises the number of important game objects that can be taken into account when making decisions. However, such a model also has drawbacks as the validity of old information in memory can quickly deteriorate. For example, in the very basic world model in which only the positions of objects are stored, as soon as an object moves, the validity of the memory of the object in its previous position is reduced. This memory may still be useful, as it does provide the player with a *vague* idea of the position of the object. The more a memorised object moves away from the position at which it was viewed, the less meaningful the memorised position becomes, as it becomes further from its true position. This effect can lead to players attempting to act

based on beliefs about their environment which vary massively from the actual state of this environment.

In order to counter this effect, some means must be taken to minimise the use of world information in players' memories which varies significantly from the actual state of the environment. To take minimising possibly invalidated memories to its most extreme would mean counting any information in memory which was not confirmed or updated in the most recent sense cycle as invalid, and so not to be acted upon. However, this would have the same effect as having players without any memory at all.

For these reasons, it makes sense to strike a balance between having a memory model which will not contain an excessive amount of misleading information, and containing enough information to be useful to players' decision-making. However, the issue of how long a piece of memorised information is useful for is contentious. For example, if a player is viewed, and does not move very far within the next 30 game cycles, the positional information memorised about this player will still be useful after this period as it will still be a good estimate of their true position. Conversely, if the viewed player were to run halfway across the pitch over this period, the memorised information would become very removed from the actual state of the world.

Following this argument, it follows for the memory model to keep information on objects for a number of cycles based on the probability of the information still being useful after that number of cycles. Here, object information would be *forgotten* when the probability of the information still being useful had deteriorated below some threshold level. This method would indeed strike a balance between retaining useful memory information and removing possibly misleading information. However, there are a number of issues which complicate this approach. Firstly, there is the problem of determining what this threshold reliability level at which to dispose with memory information should be. Set it too high, and useful memory information will be disposed of, meaning that the player cannot take advantage of the information, but set it too low, and a large quantity of misleading information could remain in the player's world model. Also, this can only be achieved with the assistance of some metric to determine the reliability level of a memory. The obvious solution is to have a reliability level which deteriorates linearly as game cycles go by. However, there is an argument that this memory reliability metric should also be dependent on the *type* of object seen (is a teammate less likely to move further than an opponent between being viewed? or is the ball more likely to move further than a player?).

Another issue with such a memory model in which information is either included or not is that, in some circumstances, a very high reliability on a piece of information in a player's world model is essential, whereas in other situations, the reliability of this information is not so important. For example, when an attacking player is judging the line of offside in order to keep in an onside position, memory information that has become slightly removed from the actual world state can lead to the player straying into an offside position, and possibly missing out on a goalscoring opportunity as a result. In contrast, such a disparity between memory and actual world state would be less important when

making a long pass to a player in space, as they will still most likely receive the pass without issue if the pass is made to a position close to them.

The strategy which I use was to not *dispose* of sense information at all. Instead, I opted for a model which allowed for some measure of *confidence* to be related to each object in a player’s world model, which could then be looked up when making decisions involving this object. This idea is in line with those used by Bowling et al. (1996) and de Boer et al. (2001), and means that decisions can be made with different levels of confidence in world model features being required for different decisions. This effectively means that players can benefit from having some world model information on any objects which have been sensed so far in a game. I chose to use the number of game cycles which had elapsed since an object was last sensed as a metric for measuring confidence. This is because, although I highlighted earlier that the reliability of memorised sense data may be dependant on other factors, I was unable to identify how the reliability of sensory information varies between different types of object.

In some cases, a memory model in which the positions of world objects are assumed to remain constant when they are not being sensed is reasonable. For example, if a player is viewed, and is stationary, there are no grounds on which to assume the player will move in any particular direction when not being viewed. Indeed, this player may move during this time, but it is very difficult to predict *where*, so given that they could move in any direction, the assumption that they will stay where they are is as good a prediction as any.

However, there are other scenarios in which it is not reasonable to assume that a viewed object will remain stationary when it is no-longer being sensed. For example, if a ball is viewed with a particular non-zero velocity, it is reasonable to assume that the ball will continue to move in the same direction when it is out of sight. A simplistic approach would be to assume that viewed objects will continue to move at the velocity which they were moving at when they were viewed – indeed, this is the assumption used in my implementation for viewed players. This approach does, however, have its drawbacks. The most obvious example for this is the ball – the motion model used in SoccerServer dictates a rate of decay at which the ball’s velocity decreases.

In order to take into account reasonable predictions to changes to the state of a player’s world model, there are two obvious choices:

- Update the world model to predict any changes to it each cycle
- Do not update the world model each cycle, but implement to allow predictions to be made of new states of particular objects when useful

The first option is intuitive in that it means that a player’s world model at any point in the game is a best approximation of the state of the world, as any predicted changes are applied to the model. However, the problem here is similar to that outlined about using a relative world model in Section 5.2.1 of having to update a large amount of information on each cycle. This is because new predictions on the state of all objects in a player’s world model would need to be calculated each cycle – a large computational overhead.

The other option is to not update the world model based on predictions of the new states of objects in it, but instead to keep the states of objects in the world model the same. Predictions of new states of particular objects can then be made for specific cycles as and when they are needed, provided the objects store the required information, such as velocity and crucially, the cycle at which they were sensed, to make these predictions. This removes the necessity of large volumes of calculations being made in each cycle, while retaining the functionality of being able to predict the states of objects when they are not sensed. However, in using this strategy, one must be careful to not implement in a way such that the same predictions are made multiple times in one cycle as a part of different decision-making processes, as this would negate the positive impact of reducing the number of prediction calculations made per cycle. Instead, one should try to cache any predictions made so that the same predictions can be used again if necessary. I opted to use this style of predictive model, as it also enabled me to utilise predictions of the state of objects for cycles other than the current cycle, which was useful when implementing actions with longer-term goals.

5.3 Skills

5.3.1 Introduction

As outlined in Section 2.5, the basic actions available to players in SoccerServer are very low-level and unrefined. For this reason, it is necessary to implement a richer set of *skills* which can be worked with, and incorporated into strategies and plans. This will avoid having to create separate implementations of very similar actions which are used in different circumstances.

Many skills detailed here are used as action primitives in behaviour modules.

Where possible, I try to differentiate *skills* which are implemented in Java code in behaviour modules from *actions* which are the low-level actions provided by SoccerServer by using these terms.

5.3.2 Turn

Although the *turn* action is one of the basic actions made available by SoccerServer, the action is not as intuitive as it could be. When executed, a *moment* must be specified, however this is not necessarily the angle through which the player will turn. The actual angle through which the player will turn depends on the player's current *inertia*, and is given by

$$\alpha = \frac{m}{1 + is}$$

where m is the specified *moment*, i is the *inertia* value as defined by SoccerServer (default is 5.0), and s is the player's speed. This means that with the default value of inertia, if a player is moving at a speed of 0.5, the action (*turn 90*) would result in the player turning through 25.7° , not 90° as one might expect.

This significant effect of inertia on players turning means that a player’s speed must be taken into account each time a *turn* is executed in order to achieve a turn through the desired angle. For this reason, I implemented a *turn* skill which could be used so that I did not have to think about this effect of inertia in every skill in which a *turn* is utilised. I was able to do this by keeping track of the player’s speed at all times through receiving *body sense* data every cycle. I stored the received body state in a *body monitor* object every time this information was received, which I could access in the implementation of skills which would be affected by it. My implementation of the *turn* skill meant that I could simply specify an angle through which I would like the player to turn, and this would be translated to a *moment* to send to the sever with the turn command in order to achieve a turn through this angle. This was calculated by

$$m = \alpha(1 + isd)$$

The moment that can be used with the turn action is only allowed to lie between a minimum and maximum moment as specified by SoccerServer, so if the moment given by the equation is not within these bounds, my *turn* skill changes it to lie within the bounds. This means that if the angle specified to turn through is not possible, the player will turn as far as possible in the right direction.

The fact that the turn skill is used in other skills means that it can be considered as a *lower level* skill than those using it. This follows the idea presented by Igarashi (1999) of having different *levels* of skills.

As a result of the asynchronous nature of visual sense information being received and actions being executed, sometimes a *turn* can be executed by a player, and then the player is able to execute another action before the next set of visual sense information is received. This means that the player’s world model is made to be inaccurate as the player is now facing a different direction to the angle stored in their world model. In order to address this, I adjusted the *turn* skill so that, when executed, it updates the player’s world model by changing their direction in the model to the predicted new direction resulting from the turn. This updating of players’ world models to take into account the effects of their actions is also used in other skills in a similar manner to in the teams described by Stone and Veloso (1998) and Amiri et al. (2004).

5.3.3 Run to ball

The very first skill to be implemented was that of running to the ball. The first iteration of this skill was very rudimentary and used the *relative* world model which was initially implemented (see Section 5.2.1). Here, the player’s world model would be interrogated to see if it contained the ball (bearing in mind that this implementation was produced before a memory model, it was often the case that a ball object was not present in a player’s world model as the ball was out of sight). If this was the case, the turn action would be executed, with the assumption that if the player turns in the same direction whenever they cannot see the ball, then they will eventually see it. In early iterations, I

used a turn with a moment of 90° . This worked on the assumption that, with a view angle of 90° , as is the default setting in SoccerServer, after three turns, the player will have viewed all of the pitch. However, as a result of the frequency of sense cycles being lower than the frequency of action cycles, a sense cycle will not occur between each turn, so this is not the case. In order to address this, I adjusted to turn through 45° , which meant that after seven turns, the whole pitch will have been viewed. In almost every case, this will mean that the ball will be viewed. The only possibility that the ball will not be viewed is in the very unlikely event that the ball moves between sections of the pitch viewed by the player as the player turns from one section to the other, with the asynchronous nature of sense and action cycles meaning that a visual sense cycle did not occur while facing the ball. If the player did have the ball in their

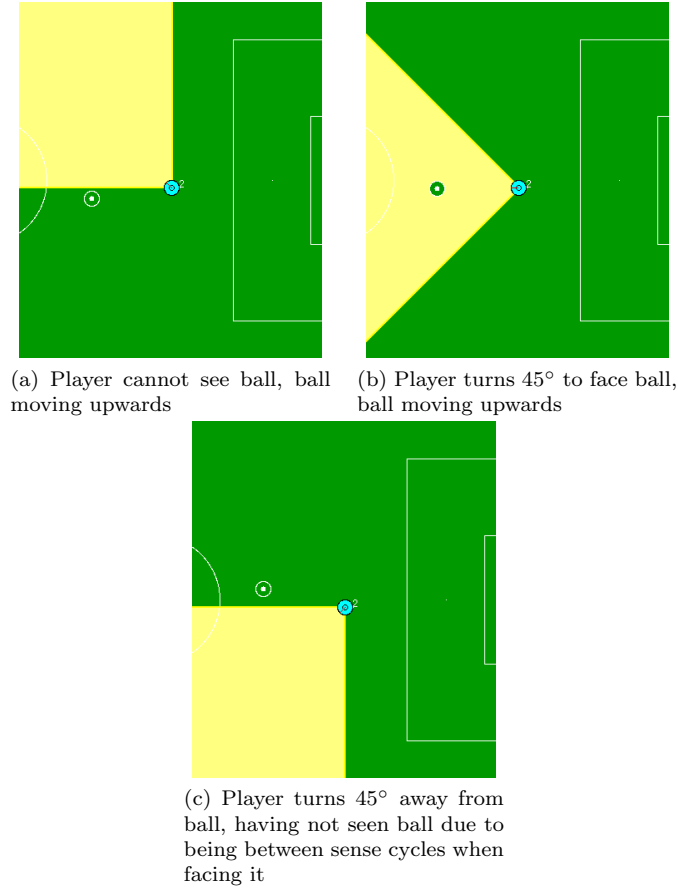


Figure 5.3: Player missing viewing ball while turning to look for it

world model and the absolute value of the relative direction of the ball from the player was lower than some threshold value (10° worked well), the dash

action would be executed with full power. This meant that when the player was facing the ball, or nearly facing it within a certain tolerance, they would run in the direction they were facing. If the absolute value of the relative direction was above the threshold value, the player would execute the turn action, with a moment equal to their relative direction from the ball. This resulted in the player turning to face the ball when they could see it and its direction was not within the angular tolerance.

Although this method did have an overall result of getting a player to run towards the ball, it was not without its drawbacks. Firstly, there was the issue of when the ball was out of sight, players could be slow to turn to face it. This would occur in the instance when the skill was set so that the player would look for the ball in a clockwise direction, and the ball is slightly out of view in an anticlockwise direction (or vice-versa). In this instance, the player would have to turn through 315° , taking seven action cycles if turning in 45° increments, where a small turn in the opposite direction would have meant that they could have found the ball much more quickly.

There was also an issue of how in some cases players could turn past the ball, then have to turn back to face it. This could occur when a player sees the ball, and turns to face it. Then in the next action cycle, as a result of no visual sensor information being received since the turn, the player's world model still says that they are not facing the ball, even though they actually are. As a result, the player repeats the same turn action in an attempt to face the ball. However, this has the result of having the player turn *past* the ball. The player would then receive visual sensor information, and see that the ball is in the opposite direction to in the previous visual sensor cycle, and so they would turn back towards it, but this time receive visual sensor information when facing the ball. The player would then be aware that they were facing the ball. In the case that the ball remained stationary, this would result in the player turning towards the ball in three action cycles, when this could have been achieved in one. However, in most cases, the ball would be moving, or another player may kick it. In such a case, depending on how the movement of the ball is synchronised with this turning past the ball behaviour, the player could be left turning in an attempt to face the ball for some time.

One approach to solve this problem would be to have players update their world model upon executing a turn action, so that the positions of all objects are taken into account. However, reasons for not choosing this approach are detailed in Section 5.2.1. Another possible solution would be to change the skill so that players only turned *half* the angle of the relative direction of the ball from them. This would mean that in cases where the player would normally turn past the ball then turn back to face the ball, they would simply turn halfway towards the ball, and then turn to face the ball, meaning that the player faces the ball after two action cycles rather than three. However, in cases in which the strategy of turning the whole angle to face the ball would have resulted in the player facing the ball after only one action cycle, four cycles would be used by turning half the believed angle as a result of slowly converging to the right direction.

Table 5.1: Actual world state and player’s world model of angle from ball when turning past it

Time (ms)	Actual ball angle	World model ball angle	Action
0			Sense
0	30	30	
0			Turn 30
0	0	30	
100			Turn 30
100	-30	30	
150			Sense
150	-30	-30	
200			Turn -30
200	0	-30	
300			Sense
300	0	0	

Table 5.2: Actual world state and player’s world model of angle from ball when turning at half angles - two cycle scenario

Time (ms)	Actual ball angle	World model ball angle	Action
0			Sense
0	90	90	
0			Turn 45
0	45	90	
100			Turn 45
100	0	90	
150			Sense
150	0	0	

Upon the implementation of an absolute world model and a memory model, I redeveloped the run to ball skill. In the first stage of this redevelopment, I used a combination of the relative and absolute world models. This worked by retaining the same process for when a player could currently see the ball – that is when the ball is in the player’s memory-less relative world model. However, I changed the manner in which the player looked for the ball when it was not. Instead of having the player turn around by set increments in a set direction, the player would turn to face the position of the ball in its absolute world model which featured memorised sensory information. This meant that if the ball was at or near to where the player believed it to be using their absolute world model, the player would turn to face it more quickly, consequently improving the efficiency of the skill in such cases. However, if the ball could not be viewed by facing the direction in which the player believed it to be, then the player may not find the ball. However, I decided that as the skill was to *run* to the ball, it did not necessarily need to involve *finding* it as well. This could be implemented

Table 5.3: Actual world state and player’s world model of angle from ball when turning at half angles - four cycle scenario

Time (ms)	Actual ball angle	World model ball angle	Action
0	90	90	
0			Turn 45
0	45	90	
50			Sense
50	45	45	
100			Turn 22.5
100	22.5	45	
200			Sense
200	22.5	22.5	
200			Turn 11.25
200	11.25	22.5	
300			Turn 11.25
300	0	22.5	
350			Sense
350	0	0	

as a separate skill.

Since this addition only improved the initial facing the ball, but worked in the same way once the ball was in sight, the problem of players turning *past the ball* was unsolved. I addressed this problem by changing the skill to use only the absolute world model. This worked because the turn skill used automatically adjusted the player’s direction in their absolute world model, meaning that the player’s direction in their world model would always be a good estimate of their actual direction. To make the skill work with the absolute world model, I added functions to calculate the relative distance and direction of the ball from the player using the data in the absolute world model. These could both be calculated using simple trigonometry. These calculated values could then be used in the same way as the relative values were used which were taken from the relative world model.

Although this implementation worked well and did result in players running towards the ball as desired, the route which players took to get to the ball was not always optimal when the ball was moving. The route taken when the ball was moving either directly towards or away from the player was optimal, as this would be a straight line. However, if the ball was moving in a direction which caused the relative angle of the ball from a player to be constantly changing, the player would take an inefficient route to the ball. This is because the player would turn to face the ball, then run towards it, but then the direction of the ball from the player would have changed, causing the player to have to turn again and run a short distance before having to turn again. This caused the player to run in a curved path, or if the ball was moving with a sufficient angular velocity relative to the player, the player would spend some time turning to track the

ball, before running to it when it had slowed sufficiently.

The effect of this is that players would take longer than the optimal amount of time to get to the ball. This is an issue in football, because as the game is played in a dynamic environment, this increases the chance for players from the opposing team to get to the ball first. For this reason it made sense to adapt the skill so that players would take an optimised route to get to the ball.

It is obvious that the most efficient route to get to any point on the pitch is in a straight line. In order for players to run to get the ball in a straight line, it is necessary to calculate where the player will be able to intercept the ball. There are a number of variables involved in where the optimal position to intercept the ball will be. These are:

- The initial position of the ball
- The initial velocity of the ball
- The initial position of the player
- The initial velocity of the player
- The direction the player is initially facing
- The player's stamina
- The player's maximum speed

In my first attempt at solving this problem I simplified the problem by making the assumption that during the process of a player running towards the ball, the velocities of the player and the ball would remain constant, with players always moving at a specified speed. This is not the actual case, as the ball slows down as it moves, and players accelerate up to their maximum speed and decelerate as they turn while moving. However, it allowed me to work with a more manageable problem which was an approximation to the real one. The solution to such an approximated problem should be an improvement to the way in which players run towards the ball over the implementation used in which the problem was not acknowledged.

Under these assumptions, I could calculate the position of the ball in n cycles time by using the equation

$$(x_n, y_n) = (x_0, y_0) + n(v_x, v_y)$$

where (x_0, y_0) is the initial position of the ball and (v_x, v_y) is the initial velocity of the ball. Also, under the assumption that the player's speed is constant, I could calculate after n cycles, a player could reach any point within a radius of ns of the player's initial position, where s is the player's speed.

Using this information, I could calculate where the player could intercept the ball in this approximated world by calculating the position of the ball at each cycle, and whether the player could reach that position by that cycle. If they could, then the position would be a possible intercept point. The optimal

intercept point could be found by performing a linear search, with the cycle being incremented by one on each pass. This would also give the cycle in which the interception would occur.

Table 5.4: Calculations for finding optimal intercept position

Cycle	Ball position	Distance from player to ball	Distance reachable by player	Player can intercept ball?
0	(15.0, 10.0)	11.4	0.0	No
1	(15.7, 10.5)	10.5	0.5	No
2	(16.4, 11.0)	9.7	1.0	No
3	(17.1, 11.5)	8.8	1.5	No
4	(17.8, 12.0)	8.0	2.0	No
5	(18.5, 12.5)	7.1	2.5	No
6	(19.2, 13.0)	6.2	3.0	No
7	(19.9, 13.5)	5.4	3.5	No
8	(20.6, 14.0)	4.5	4.0	No
9	(21.3, 14.5)	3.7	4.5	Yes

In this model where the ball’s velocity remains constant, the intercept position given by this calculation will be ahead of where the ball will actually be at the cycle in which the interception was anticipated as a result of the actual deceleration of the ball. However, if the calculation is performed each cycle, the ball’s new velocity will be used in each calculation, and so the calculated intercept position will converge to a position at which the player can actually intercept the ball as the player approaches it and the ball slows.

Although this works in the case when an intercept point can be found, there are situations in which no intercept point can be found using this calculation. This occurs when the ball is moving at a velocity at which it is moving away from the player at a speed greater than the speed of the player as used in the calculation. In some cases, this would mean that players would not be able to calculate a position at which to intercept the ball, when with the actual motion of the ball, the player would be able to intercept the ball as a result of how the speed of the ball *decays* over time.

In order to address this, I adjusted the skill so that the player would attempt to calculate an intercept point, then if no intercept point could be found within a maximum number of cycles, the player would run directly towards the ball, as if it were stationary. This meant that the player would run towards the ball before an intercept point could be calculated, then after the ball had slowed sufficiently, an intercept point would be calculable. It also meant that the search for an intercept point would not continue indefinitely when one could not be found. However, this still meant that players would still exhibit a *curved path* when running towards the ball.

In order to improve this, I used a more complex approximation of the ball’s motion which took into account the decay of the ball’s speed. According the motion model of the ball in SoccerServer, the ball’s velocity in the next cycle

will be its current velocity multiplied by a decay factor. This means that the position of the ball in n cycles' time can be calculated by

$$p_n = p_0 + v_1 + v_2 + \dots + v_{n-1} + v_n$$

where p_0 is the player's initial position, and the v_i s are the velocities of the ball in i cycles' time, which can be calculated recursively by

$$v_{i+1} = dv_i$$

where d is the rate of decay of its speed. However, this means that to calculate the ball's position after n cycles, the sequence (v_i) must be calculated up to v_n , resulting in the calculation being very computationally expensive when calculating for in many cycles' time. Also, having to calculate the ball's position at every cycle until a position that it can be intercepted at is in itself an expensive calculation.

In order to reduce the cost of this calculation, I adjusted it so that it would calculate the ball's position every cycle for the first 10 cycles, then every 5 cycles up to 30 cycles' time, and every 10 cycles up to 50 cycles' time. This was a reasonable simplification, as the further in the future for which an intercept point is calculated, the slower the ball will be moving, and so the less the effect of a few cycles would have on the ball's position. However, this still meant that the sequence (v_i) would have to be calculated up to the cycle where the player would intercept the ball. In order to remove the necessity of doing this, I used an approximation of the ball's motion in which its speed deteriorated smoothly instead of on a cycle-by-cycle basis. This allowed me to calculate a predicted position for the ball at any given cycle without calculating the sequence (v_i) . Using this smooth degradation of speed, the ball's position in n cycles' time can be calculated by

$$p_n = p_0 + \frac{v_0(d^n - 1)}{\log(d)}$$

Although this did not exactly model the motion of the ball, it was a close enough approximation to calculate where the player could intercept the ball to be effective in the *run to ball* skill.

The calculation of where a player could intercept the ball still used the assumption that the player's speed would be constant. For this reason, it was important to make a good choice of speed at which to assume the player moves at. The player's maximum speed could not be used, as if the player has to turn towards the ball, then they cannot be moving at full speed while moving. A speed too close to the maximum speed could not be chosen either, as then if the player had to turn too much, they would attempt to intercept the ball at a position which they could not reach in time. For this reason, it was better to assume a speed below the actual average speed of a player while running towards the ball, as then they would run to where the ball would go ahead of time, rather than running to where the ball had *been*, not reaching it in time. I experimented with a range of values for this speed, and settled with a value of

half of the player’s maximum speed as a good choice. However, this assumption is less effective when the player is already close to the ball, and is stationary, as over the time taken for them to reach the ball, their average speed may be below half of their maximum speed, as the majority of this time would be spent turning and accelerating rather than moving at speed. For this reason, the calculations made for where players should be able to intercept the ball may be improved by using a better approximation of the player’s average speed when moving towards the ball.

Calculations of where players should intercept a moving ball could also be improved by using approximations of its motion closer to its actual motion, but as discussed, this would be more computationally expensive. However this may be acceptable when reducing the number of cycles per second in SoccerServer, by running on a machine with more processing power, or by running players on separate machines. These better approximations would reduce the amount of time players have to spend turning as the result of new approximate intercept points calculated during running to the ball which differed from previous approximations by less. This would result in the player having a higher average speed when moving to the ball, meaning that they would reach the ball more quickly.

5.3.4 Go to

In order to easily get players to move to specific positions on the pitch, I implemented the *go to* skill. This allowed me to specify a coordinate on the pitch, and a player would use the *turn* skill and the *dash* action to get to this coordinate. This skill was implemented in exactly the same way as the *run to ball* skill in the case that the ball is stationary, but using the given coordinates instead of those of the ball; in fact, the *run to ball* skill uses the *go to* skill as a library function.

5.3.5 Shoot

My initial implementation of the *shoot* skill used the relative world model, as implemented during early development. This implementation of the skill worked by checking if the opposition’s goal was in the player’s world model – meaning that it was viewed in the most recent set of visual sense data. If the goal was in the world model, then the player would *kick* with maximum power at the relative angle of the goal from the player, as contained in their world model. In the case that the player’s world model was synchronised with the actual state of the world, this resulted in the player kicking the ball with full power towards the goal. However, in situations in which the world model of the player was not an accurate representation of the world as a result of some action being executed between visual sense cycles, this would not necessarily be the case. For example, if a player had turned 90° between visual sense cycles, the *shoot* skill would kick the ball at full power at 90° from the goal. This occurred quite

regularly, resulting in players kicking the ball out for a throw in or wide of the goal at full power.

A second iteration of the skill was implemented using the absolute world model. This version of the skill worked in essentially the same way, with players kicking the ball at full power at the relative angle of the goal from the player. However, as the position of the goal is fixed, it was always present in players' world models, which meant that they could always attempt a shot when they could kick the ball, even if they could not see the goal. Also, as later iterations of the *turn* skill took into account the effects of the turn and updated the world model accordingly, this eliminated the problem of shooting wide of the goal. Player movement between visual sense cycles was still not taken into account in the world model, but this did not have a negative effect on shooting, as the player's movement in one cycle would have a negligible effect on the relative direction of the goal from the player.

The skill meant that the player always shot towards the goal. However, because the *goal* object, as received in visual sense data, is positioned at the centre of the goal and because the a player always shoots in the direction of this object, players will always aim at the centre of the goal. Although this means that the likelihood of the shot being on target is maximised, it also does not make for the most difficult shots for the opposing goalkeeper to catch, assuming that they position themselves in the centre of the goal. Noise is applied to kicks, so the result of any shot will not necessarily be precisely to the centre of the goal, and also the goalkeeper will not always be positioned in the centre of the goal, so players may still score with the *shoot* skill.

In order to improve the *shoot* skill, I could adjust it so that players would shoot towards the point on the goal line furthest away from the goalkeeper. This would minimise the chance of the goalkeeper being able to catch the shot.

5.3.6 Clear

I decided to implement a *clear* skill as a way for players to quickly get the ball away from their own goal in order to stop the opposition from scoring. This essentially means kicking the ball hard up the field away from their own goal.

My first iteration of this skill was implemented using the relative world model. This worked in a very similar manner to how the *shoot* skill worked in the relative world model, but the player would instead see if their *own* goal was in thier world model, and if it was they would kick the ball at full power at 180° from the relative direction of thier own goal. This resulted in the player kicking the ball in the most direct route away from their own goal. Of course, the same issue with world model information differing from the actual world state, as present in the *shoot* skill using the relative world model, causing the kick to be executed in a direction other than that desired also existed here.

Although the skill used with the relative world model was successful in cases where the player could see their own goal, it did not work in other situations. Also, although the skill meant that the ball was kicked towards a position which was "safer" for the defending team, it often meant that the ball was kicked off

the pitch, resulting in a throw in being awarded to the opposing team, so giving away possession.

I addressed the issue of when the player cannot see the ball by using the absolute world model instead of the relative model. The problem of kicking the ball directly off the pitch was solved by removing the need to use a separate *clear* skill at all, and instead using the *shoot* skill instead in its place, as this also kicks the ball away from a player's own goal with full power.

5.3.7 Dribble

I implemented a skill for players to dribble the ball as a means of advancing it up the field while retaining possession. The actual dribble *skill* that I implemented does not actually prescribe how a player should act during the entire process of dribbling. Rather, it is simply the *kick* part of the dribble, with the running after the ball part being implemented by including in plans alongside the *run to ball* skill.

I initially implemented this effectively in exactly the same way as the *shoot* skill, but with a much lower power of kick. This meant that all dribbling would be directly towards the opponents' goal. However, when coupled with the *run to ball* skill, players started sometimes exhibiting strange behaviour when dribbling, where they would kick the ball a small distance as expected, then turn around and face away from the ball. This was as a result of the ball's velocity being changed as a result of the kick, and this new velocity not being picked up by a visual sense before the next action cycle. During this time, the ball in the player's world model would have been slowing down while the player was moving in the direction of the ball. As a result of the velocity of the ball in the player's world model not being updated as they kicked the ball, and from the predictions made in the player's world model based on their last sensed velocities of the ball and themselves, the player believed that the ball was behind them, and so they turned around.

I corrected this problem by implementing *kick* as a skill which used the *kick* action. When this new *kick* skill was executed, the ball's velocity in the player's world model would be updated to the expected resultant velocity from the kick. According to the model used in SoccerServer, the resultant velocity of a kick is calculated by

$$v = u + a$$

Where u is the previous velocity, v is the new velocity, and a is the acceleration applied to the ball. The effects of the random noise on motion and the wind are then applied to this velocity, and the velocity is normalised to be at most the maximum ball speed as defined by SoccerServer. The acceleration a is given by the sum of all *power* vectors of kicks applied to the ball in the cycle – multiple players can kick the ball in the same cycle. This is then normalised to be at most the maximum ball acceleration as defined by SoccerServer. The resultant

power of a kick is given by

$$p = e(1 - 0.25\frac{d}{180} - 0.25\frac{s}{m})$$

where e is the power with which the player attempted to kick the ball, d is the absolute value of the relative angle of the ball from the player, s is the distance from the player to the ball, and m is the *kickable margin* parameter supplied by SoccerServer which defines how close to the ball a player must be in order to kick it.

The predictions I made for the ball's new velocity after a kick were calculated using the same process as SoccerServer uses to determine the ball's velocity, apart from two simplifications:

- Noise is not taken into account, as this cannot be predicted
- Only the resultant power of that particular player's kick is taken into account, and it is assumed that no other player kicks the ball, as this cannot be predicted

By predicting the new velocity using a process as closely aligned to the process through which the ball's velocity is actually calculated, the prediction can be as accurate as possible.

When this predictive element was applied to *kick*, the problem of players believing that the ball was where it would have been had it not been kicked was solved. This meant that players could kick the ball and immediately run to where they had kicked it, as they could accurately predict where it would be without having to receive more visual sense data after the kick.

Although the *dribble* skill worked well in most circumstances for a player to dribble directly towards the opponent's goal, when receiving the ball at pace, the skill was not so successful. This is because in the dribble skill, players always kick the ball with an attempted kick power of ten. This means that as a result of the kick model in SoccerServer, depending on the exact position and velocity of the ball, this would result in different velocities of the ball. For example, if the ball was travelling very fast towards a player in the opposite direction to the opposition's goal, a player's first kick in an attempt to dribble the ball would simply slow it down, rather than start it moving towards the opposition's goal.

This problem could be solved by implementating a *kick* skill in which rather than specifying a direction and the power for a player to kick the ball with when executing, I could specify a desired resultant velocity of the kick in a similar manner to that described in Yang et al. (2002). In order to do this, the skill would need to calculate the attempted power and direction to apply to the ball in a given situation in order to achieve the desired velocity, and then execute the *kick* action with these parameters.

The skill which I implemented makes players dribble *directly* towards the opponent's goal. Although this is a good way to advance up the field, it is not the safest way to retain possession of the ball. For example, the skill may mean that players sometimes dribble the ball directly into an opponent. The

skill could be improved by making players dribble away from opposing players while still advancing up the pitch. This could be done by writing a function which would calculate a utility value for dribbling in a number of directions based on the field position and proximity from opposing players of the target location. The direction with the highest utility could then be selected for the player to dribble in. The biases on field position and proximity to opposing players in calculating this utility value could be adjusted in order to achieve the best results.

5.3.8 Pass

In implementing a *pass* skill, several decisions had to be made. These included answering questions such as should the skill decide who to pass to, or should this be determined by another skill? and what criteria should be used when deciding who to pass to?

In the approach that I opted for, the skill did involve the process of selecting the player to pass to. This worked by assessing the teammates in a player's world model and selecting the player in the best position to pass to. This decision also took into account the confidence in the player's position in the world model, as attempting to pass to a player with a low confidence in that player's position means that there is a high chance that the player will be far from that position, and so they would not receive the ball and possession would be lost. To achieve this, only players in the world model which had been sensed within a set number of cycles would be considered as pass targets. As a result of players only being added to the world model when their jersey number is sensed, these players would always be within a reasonable distance of the player making the pass. From these players, the player in the most advanced pitch position would be selected as the pass target.

The actual execution of the pass to the target player involved kicking the ball in the relative direction of the target player from the player making the pass with power relative to the distance of the target from the passer. The decision of the power to put on the pass is important, as if hit too powerfully, the receiver must be exactly in position to receive the pass in order to stop it from going past them, and if it is too weak, it will not reach the intended receiver. After some experimentation, I settled on a value for the power given by $p = 2d$, where d is the distance from the passer to the receiver, as a good choice of power.

Although the actual passes made were reasonably accurate, I encountered problems with players *receiving* passes. These problems were generally that the receiver was not looking towards the ball when the pass was made, meaning that they were unaware that the ball was being passed to them. I identified two ways to solve this problem. The first was to ensure that players looked at the ball more frequently, thus increasing their likelihood of seeing a pass made to them. The other was to employ some *communication* between players in order to aid passing.

The option of having players look at the ball more frequently suffered from the fact that if players were to spend more time looking at the ball, they would

have less time in which to position themselves. For this reason, I disregarded this option. As for using communication to help passing, I thought of two main methods:

- Have the player looking to receive a pass communicate to the passer that they would like to receive a pass
- Have the player making the pass alert its intended recipient of the pass

The choice to have potential pass receivers *call for* passes would mean that the player calling for a pass could call for it then enter some sort of *receive pass* state in which they would look at the ball, ready to receive a pass. However, if communication were to be only from the potential receiver to the player with the ball, calling for a pass would not necessarily mean that a pass would be made, so a player may enter a *receive pass* state when other action may be more appropriate. It is for this reason that I chose to use the approach of having a player *announce* that they are making a pass.

The strategy I used here was simply for the passer to announce the position of the ball when making a pass, then any potential receivers could add the ball at the announced position to their world model if they were not already looking at it. This uses the same idea as de Boer et al. (2001) whereby communication between players is used to augment their world models. The main challenge that I faced in implementing this was the character limit imposed by SoccerServer on *say* actions – ten characters. Ideally, I would have liked the passer to announce the ball’s location, the velocity it was to be passed at and the jersey number of the intended receiver in order for teammates to be given a fuller picture of the passer’s intentions. However, with such a character limit in place, I saw no way of communications providing this depth of information. With the potential receiver being aware of the position of the ball at the point of passing, they were far more likely to look at it while the ball was heading towards them, and so successfully receive the pass.

As with the *dribble* skill, the kicks executed in passing also suffer in cases in which the ball has a significant velocity at the time of kicking. Again, this could be solved by implementing a *kick* skill in which the desired resultant velocity can be specified when executing it.

Another area in which my *pass* skill could be improved is in the selection of where to aim to passes to a particular player. In my implementation, passes are played *directly* to a player. It may be advantageous for passes to sometimes not be played directly to players, but *in front* of them slightly so that the receiver can move to this more advantageous field position to receive the ball while the ball is travelling to it.

Also, in my implementation, the selection of a pass target only takes into account the positions of teammates – it does not take into account the positions of opposing players. This means that in some instances, passes are made which can be easily intercepted by opposing players. Also, sometimes the teammate to whom the ball is passed is very close to an opposing player who will have a good chance of getting the ball after the pass has been completed. This situation

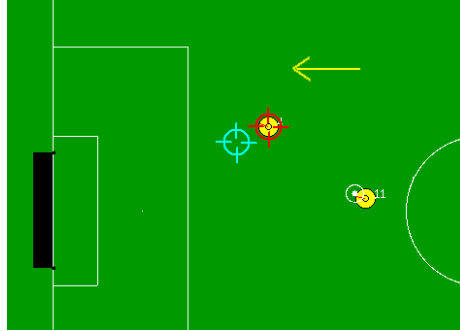


Figure 5.4: Pass targets directly to receiver and slightly ahead of receiver

could be improved upon by taking into account the positions of opposition players when choosing a pass target. This could be incorporated into the idea of using a utility function by having the utility function take into account the proximities of the nearest opposing player from the line of the pass and from the prospective pass receiver. This would potentially reduce the number of passes which are intercepted, while still allowing more “risky” passes to be made if there is sufficient advantage to be gained from making such a pass.

5.3.9 Go To Defensive Position

When defending, there are a number of factors that should be taken into consideration when deciding where on the pitch a player should be positioned. Their position must complement the positioning of teammates, and should be based on the specific situation that the player is in. A very simple strategy would be to assign all players a set *defensive position* which they should go to whenever they are defending, only leaving this position when certain criteria are satisfied. However, this approach is very inflexible, and would not work well in all situations. For example, if the opposing team was advancing up the pitch, and the ball was closer to a particular defender’s goal than that defender was, then the defender would no longer pose any resistance to the attack.

The approach which I opted for was also very simple, but was more *reactive* to the situation in hand. This was for players to simply maintain a constant lateral position, but with a distance up the field equal to the ball’s distance up the field minus a “buffer” distance, but no further up the pitch than the most advanced point of the player’s positional rectangle (as described in Section 5.4.7). This meant that players would keep defensive positions covering the whole width of the pitch (assuming the positions allocated to the players did), and players would not stay up field, offering no defensive help. It also meant that defenders would keep a *defensive line* as in the team described by Nakagawa et al. (2000), which meant that opposing players could not wait around too far up the pitch without being offside.

5.3.10 Stand Up

When defending, it is sometimes a good decision for a player to position themselves somewhere which would make it difficult for the opposing player with the ball to advance past them or shoot. It is for these situations that I chose to implement the *stand up* skill. This skill makes the defending player position themselves in a direct line between the ball and their own goal at a set distance from the ball.

5.3.11 Go To Attacking Position

I implemented a skill for players to get into an attacking position in a similar way to the *go to defensive position* skill. This meant that according to this skill, players would maintain the same position across the width of the pitch, but take a position up the field related to their positional rectangle, the positions of opposing players and the position of the ball. The distance up the field selected for this attacking position was the *offside line*, which the player calculated as the furthest distance up the field they could get without being in an offside position, minus some buffer distance. The buffer distance was used to ensure that players would not be caught marginally offside as a result of minor inaccuracies in their world models. This position was capped at the most advanced point of the player's positional rectangle. This meant that players would attempt to remain in an onside position, and that not all players would advance too far up the pitch.

5.4 Percepts

5.4.1 Introduction

In addition to the implementation of the *skills* described in the previous section which resulted in a player in executing a specific action, I also implemented a number of *percepts* which are functions for interpreting a player's world model in some way. Similarly to how many of the *skills* were implemented as action primitives in behaviour modules, many of these *percepts* are implemented as sense primitives in behaviour modules.

5.4.2 Can Move

SoccerServer allows players to use the *move* action to instantly move to any position on the pitch before kick off at the start of a half or after a goal has been scored. For this reason, I implemented the *can move* percept to check if the *move* action is legal at that point in the game.

Any change in *game state* is announced by the referee, which all players will receive. In order to keep track of the current game state, players store the last announced game state. This is then used with the *can move* percept; the result is based on whether the current game state is *before kick off*.

5.4.3 Is Dead Ball

It is often useful for players to behave differently in dead ball situations such as free kicks and throw ins to in open play. For this reason, I implemented the *is dead ball* percept which simply returns whether the game is currently in such a state. The percept works in the same way as the *can move* percept – by checking if the current *game state* is a dead ball state.

I also implemented an *is our dead ball* percept which returned if the game was currently in a dead ball state which the player’s team was to restart play from.

5.4.4 Can Kick Ball

I implemented a *can kick ball* percept to enable a players to make judgements on whether they are in a position to kick the ball in the next action cycle. This meant calculating if the ball is close enough to the player in the next cycle to kick the ball. This is dependent on the *kickable margin* and the *ball size* as determined by SoccerServer, and also the *player size* which varies with heterogeneous players. These variables define the maximum distance that a particular player must be from the ball in order to be able to kick it, given by $maximum_kick_distance = kickable_margin + ball_size + player_size$.

The result of the percept is calculated by comparing the distances between the player and the ball in the player’s world model. The positions used for these objects in the world model are those for the *next* action cycle, so any movement in the current cycle is taken into account based on the current velocities of the player and the ball. This avoids players believing they are in a position in which they can kick the ball, when in fact, this was only true for the previous cycle.

Although this percept is successful at determining if a player is able to kick the ball, it gives no indication as to what the *effective power* would be on a kick from the position. This varies with the player’s distance from the ball, and the direction of the ball’s location relative to the player. Such information may be useful when making some decisions, such as when to shoot, as it may be only desirable to use such an action if a certain power can be applied to the kick. For this reason, it may be beneficial to extend the percept to return the *maximum effective power* on a kick from the player’s location, rather than simply returning a boolean value of whether they are *able* to kick the ball.

5.4.5 Can Catch Ball

The can catch ball percept is used by players to decide if they are in a position where they can catch the ball. Obviously, in football, only the goalkeeper is allowed to catch the ball, so for any player other than a goalkeeper, the percept will always return as *false*. However, for goalkeepers, the percept works in a very similar manner to the *can kick ball* percept, apart from it will always return *false* if according to the goalkeeper’s world model, they are outside of their penalty area.

5.4.6 Is Ball Location Known

Many decisions that players make while playing football are based on where the ball is. For this reason, it is important that the players maintain an understanding of the location of the ball. I chose to implement the *is ball location known* percept so that players could make their decisions based on whether they were actually aware of where the ball was.

My first iteration of the percept worked by checking a player's world model and returning whether the ball was sensed in the player's most recent visual sense cycle. This meant that the player had the highest confidence that they could possibly have at that point of the ball's location. However, this led to actions in plans which were only triggered if a player knew the ball's location being executed very rarely, especially when players were trying to get to a position for which they had to run *away from* the ball to get to. For this reason, I refined the percept to instead use a threshold value for how recently the player had seen the ball. This worked well, as if a player had seen the ball very recently and a predictive memory model is used, the predicted location of the ball can be used with a high level of confidence.

This implementation worked as a reference when making decisions, however there are some decisions for which more or less confidence in the ball's location is required when making. For example, an attacking player running to a position to make themselves available for passes requires less confidence in the ball's location than when a player is trying to shoot. In order to make these decisions, it may be useful to adjust the percept to return a value representing the player's confidence in the ball's location, which can then be considered when making decisions, rather than a boolean value representing if the ball's location is *known* or not.

5.4.7 Is In Position

The *is in position* percept is an interpretation of a player's position on the field which determines whether that position fits in with the player's role in the team. For example, if a player takes the role of a central defender within the team, they would be considered to be *out of position* if they were in the opponent's six yard box. Such information can be useful when assessing whether action should be taken by a player in an effort to maintain their team's overall *shape*.

In order to assess whether a player is *in position*, it is necessary to define *where* on the pitch they could be that they should be considered as being in position. This decision can be made in a number of ways, and depends on how a player's *position* is defined.

In my first iteration of the percept, similarly to the implementation described by Stone and Veloso (1998), I assigned players a *home* coordinate, which could be given in an *init* file. This *home* coordinate would form the basis of all of a player's positional decisions. In this initial implementation, a *freedom* value was set, meaning that a player was considered to be in position as long as their distance from their *home* coordinate was at most this *freedom* value. This

worked well in terms of keeping the team's shape, however, there were a number of drawbacks to the approach.

One such drawback is that if players' home positions were defined to be the positions that they assumed at the start of the game, which must necessarily be inside their own half of the pitch, the players' *freedom* must be very high in order for them to reach advanced positions inside the opposing half. For example, a centre forward may assume a position near to the centre spot. There are situations in which it makes strategic sense for the player to be as far up the pitch as the opponent's goal line, such as from an attacking corner. This means that if this position is to be considered as *in position* for this player, then this must apply for almost everywhere on the pitch. This does not necessarily represent the true area on the pitch for which this player should be considered to be *in position*, as in most circumstances, a centre forward would be considered *out of position* if they are on their own goal line.

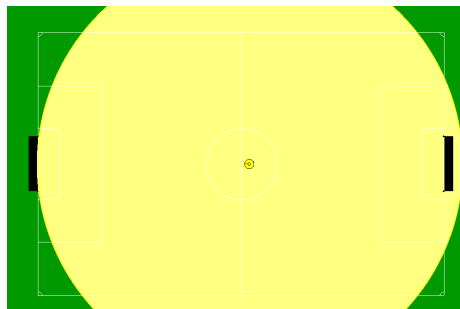


Figure 5.5: Potential area a centre forward considers to be in position using *home position/freedom* positioning approach

This issue could be addressed by making a player's *starting* position (their position on the field at a kick off), independent from their *home* position (where the area of the pitch in which they are considered *in position is centred*). This would allow these *in position* areas to be smaller, potentially resulting in the team keeping a more rigid shape.

After using this model for player positioning for some time, it emerged that it did not always make sense for the areas in which a player should consider themselves to be *in position* to be circular. This was because sometimes it made sense for players in particular positions to have more or less freedom from side to side on the pitch than in moving up and down the pitch. For example, it may be useful for a wide midfield player to have a large range from one end of the pitch to the other, but should remain constrained to wide positions. For this reason, I decided to replace the circular positional areas with rectangular ones like as described by Wang et al. (2008), thus allowing such options to be used.

This approach also made it easier to allocate positions for all players on the team to ensure that any point on the pitch was considered to be *in position* for

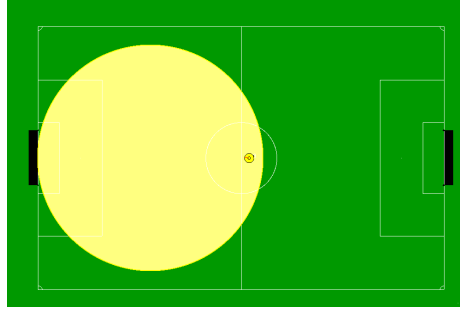


Figure 5.6: Potential area a centre forward considers to be in position using *starting position/home position/freedom* positioning approach

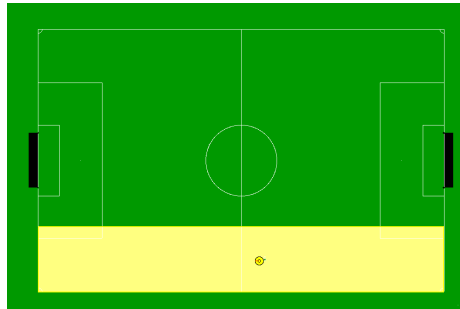


Figure 5.7: Potential area a left midfielder considers to be in position using *rectangular* positioning approach

some player. This meant if the ball were at any point on the pitch, at least one player on the team could get it while remaining *in position*. This was more difficult to do with circular areas, while keeping reasonable *in position* areas for certain positions.

Using the rectangular approach, it was easy to check if a player is *in position* by comparing the player's coordinates with the bounds of their *positional rectangle*, and checking if they lay within the bounds.

5.4.8 Is Too Close

In the majority of situations, it is not desirable for a player to be too close to a teammate. When attacking, this is because passing to a player very close by gives little advantage as it will neither result in getting the ball into a significantly better position, nor move it significantly further away from opposing players. When defending, it means that both of the players will be effective against attacks in the same positions, when they could be better positioned to further reduce attacking options. Because of this, it is useful for players to have some percept at their disposal to assess whether they are too close to a

teammate in order to determine when they need to resolve such a situation.

The percept was implemented by assessing a player's distance from all teammates in their world model, and returning *true* if any of these distances were below a threshold value. The choice of this threshold value was an interesting decision. If a value is chosen which is too high, players may avoid positioning themselves in a potentially advantageous position as a result of a teammate being too close. This could occur when the defending while the ball is in a player's own penalty area, and so it could make sense for many defenders to be in the penalty area. If a value is chosen that is too low, this may result in players becoming too close to each other.

I experimented with a range of different values for this threshold before settling with one based on the relative effects of the trialled values that I perceived while watching simulated games. Although the value that I settled with worked well in many situations, it became apparent that in different scenarios, it is more or less important for players to avoid being too close to teammates. For this reason, I believe that this percept could be improved by implementing some utility function to choose an appropriate threshold value based on the situation. This should probably take into account field position, whether the team is currently attacking or defending, and the players' proximity to opposing players.

5.4.9 Is Attacking

In football, the decisions a player makes are largely based on whether their team is currently in possession of the ball. In order to make these decisions, it is necessary to have a way of determining which team is in possession of the ball. For this reason, I chose to implement a percept to check if a player's team is in possession of the ball.

When the ball is at a player's feet, it is obvious that the team to which that player belongs is in possession of the ball. However, when the ball is not very close to a particular player, the team that should be interpreted as being *in possession* for decision making purposes is not such a simple observation to make. There are a number of possible interpretations in these cases, which include:

- The team of the player closest to the ball
- The team of the player to touch the ball most recently
- The team of the player closest to where the ball is travelling to

Each of these interpretations has its advantages and disadvantages.

Taking the team in possession of the ball to be that of the player closest to the ball is the easiest to interpret from a player's world model. However, this choice does not always accurately represent which team has control of the ball. This can be because the ball could simply be travelling past a player quite near to them while being passed between opposing players, but not close enough for the player to be able to intercept it. In this situation, the player that the

ball was travelling close to would be interpreted to be *in possession* of the ball, although they have no control over it.

Interpreting the team of the player to most recently touch the ball as being in possession makes sense in most situations. For example, if a player is dribbling the ball, they will be closest to it, also if the ball is passed to a teammate, it makes sense for the team making the pass to be considered as in possession while the ball is travelling to the receiver. However, in other cases, such as if a player makes a poor pass, clearance or shot, for which the opposing team will gain possession, the opposing team will not be considered to be *in possession* until one of its players has touched the ball. This may not be desirable, as the team losing possession of the ball may benefit from being able to react to losing possession more quickly if they were to know they had lost possession before their opponents touched the ball. Also, this approach would involve implementing a method for determining when a player touches the ball, and which player touched the ball.

Interpreting the player in possession of the ball as being the player closest to the ball's destination means that when passes are made, the team making the pass can continue to be believed to be in possession. However, the approach would not take the possibility of balls being intercepted into account.

None of the approaches address the problem of player information not being present in a player's world model as a result of full identity information not being received for the player. This means that only players for which the team to which they belong is present in a player's world model can be considered when deciding which team is in possession.

I chose to use the approach of considering the team of the player closest to the ball at any given time to be in possession. This was both for simplicity and because the amount time in which a misleading conclusion could be reached from the method is relatively small. However, the percept could be improved by combining with the other methods in some way in order to minimise the effects of the problems apparent with any one of the methods.

5.4.10 Is Pass Available

When making the decision to make a pass, it will normally be useful to first decide whether there is a player *available* to pass to. For this reason, I implemented the *is pass available* percept. The percept worked using the decision making part of the *pass* skill, and returning *true* if it was able to determine a pass target, and *false* otherwise. The percept meant that plans could be written in which players would only make passes if they had an intended target for one.

The percept could be further developed to return a numeric value representing either the confidence that a pass would be successful, the positional advantage that would be gained by making a pass or a combination of the two. This could improve the scope for decision making on whether to make passes based on the risk involved in the pass or on how effective the pass could be.

5.4.11 Is In Shooting Range

I implemented the *is in shooting range* percept with the sole purpose of giving players a way of deciding when to shoot. My implementation of this worked, as the name I have used for it suggests, simply by checking the player's distance from the opposing goal. If this is below a certain threshold distance, the player is judged to be within shooting distance.

Although my implementation does work as *a* method of deciding whether a player should shoot, there are a number of other criteria which could be taken into account when making this decision. One such criterion is the position of teammates. This is because shooting may not be the best choice if a teammate is in a position from which a shot is more likely to be successful, and this teammate is open to receive a pass. Another consideration when deciding to shoot is the position of the opposing goalkeeper. This is because it may be a good choice to shoot from further than would normally be considered if the goalkeeper is out of position. Similarly, the positions of opposition defenders could also be considered when deciding whether to shoot because if there is a high probability of a shot being blocked, it may be a poor decision to shoot.

I also implemented a percept *is near own goal* with the primary intention of detecting when urgent defensive action is required, such as clearing the ball. This was implemented in exactly the same manner as the *is in shooting range* percept, apart from the goal the player was defending and a different threshold distance were used.

5.4.12 Is My Nearest

Many decisions players can make during the course of a game of football benefit from considering the positions of the player's teammates. In particular, it is often useful to consider if a player is the closest player on their team to the ball when deciding what to do. Taking this into consideration can help to avoid multiple players on the team running to the ball at the same time, rather than having one player run to the ball, and the others attempt to position themselves to be available for passes, or to defensive positions. For this reason, I implemented the *is my nearest* percept which simply returned whether the player believed that they were the closest player on their team to the ball based on their world model.

The percept does have the limitation that it relies on the player's world model. With this, problems with not having complete identity information on players are brought into the percept. This means that sometimes the result of the percept may not always relate to the actual state of the world as a result of players' positions being disregarded because of identity information not being available for these players. This could mean that players believe that they are the closest on their team to the ball when they are not. This could potentially lead to scenarios where multiple players are chase the ball. However, in the majority of cases where this could occur, the players would see their teammates involved in the situation, and after they had become close enough, they would

receive identity information on them, and the situation would resolve itself.

5.5 Plans

The plans which I created throughout development grew in complexity as the process continued. This was to be expected, as in early iterations, plans were very simple. For example, the very first plan which I implemented meant that players would shoot whenever they were close enough to the ball, otherwise run to or look for it. However, later plans were much more complex with many elements. Samples of such complex plans for both outfield players and goalkeepers can be found in Appendix A.

5.6 Development Process

5.7 Initial Stage

I tried to follow the development process as defined by BOD as closely as possible. However, before considering the behaviour of agents at all, I decided to implement an interface in Java which could interact with SoccerServer. This had to deal with connecting to the server and providing methods for sending actions to the server. I also implemented a parser to parse received visual sense information and store the set of most recently viewed set of objects. This formed the relative world model with memory of only the most recently received set of visual sense data. I then had an interface to interact with the server and had a source from which to derive senses. This formed a basis on which to implement behaviours.

5.7.1 Initial Decomposition

Before implementing behaviours, I made an *initial decomposition*, as specified for BOD, in order to start to identify the behaviours which would need to be implemented. This initial decomposition was based on a very basic intended agent. This very basic agent would simply run towards the ball, and shoot when at the ball.

In this initial decomposition, there were three behaviour modules:

- Player
- Vision
- Movement
- Kicking

The *player* behaviour module dealt with an agent's interactions with the server, such as timing when actions should be executed and providing an interface with the Java code in which any state was encapsulated. The action primitives provided by the player behaviour module were *connect* and *idle* which connected to SoccerServer and did nothing respectively. The sense primitives supported by the behaviour module were *is_connected*, *can_send_command* and *is_alive*. The player behaviour module had a very different role to the other behaviour modules, as it existed as a means to enable the other behaviour modules to work, while the other behaviour modules were independent of each other.

The *vision* behaviour module supported any sense primitives arising from a player's visual sensors. In the initial decomposition, the behaviour module supported the *can_see_ball* and *can_kick_ball* sense primitives and no action primitives.

The *movement* behaviour module was used for all action primitives concerning movement of a player. In the initial decomposition, this supported the *run_to_ball* and *turn* action primitives. This behaviour module supported no sense primitives.

The *kicking* behaviour module supported any kicking action primitives. In the initial decomposition, this consisted of only *shoot*. The kicking behaviour module did not support any sense primitives.

5.7.2 Iterative Design

From this decomposition I implemented one behaviour at a time, and incorporated the behaviour into a very simple plan in which it was used in order to test the behaviour. When all the identified behaviours were implemented, I constructed plans to use these behaviours so that an agent could perform at a high-level as intended. I then viewed the agents using the plans, and analysed their overall performances. From this I identified how these performances could be improved by either altering existing behaviours, implementing new behaviours to be incorporated into plans or by changing the plans used using existing behaviours.

If this analysis resulted in a conclusion where a new behaviour were to be implemented or an existing behaviour were to be altered, I carried out another decomposition. This process was carried out iteratively, and frequently resulted in behaviours being altered, and in some cases completely rewritten. Each time an alteration to a behaviour was made, I would test the behaviour as independently as possible. In some cases, testing a behaviour completely independently was not possible, and had to be implemented in some plan in which it was complemented by another behaviour or behaviours in order to test. For example, testing various ways of kicking the ball would often be tested in conjunction with behaviours which allowed the players to get to the ball to enable them to kick it.

5.7.3 Final Decomposition

Following the iterative design process outlined in the previous section, the final behaviour decomposition featured percepts as described by Table 5.5.

Table 5.5: Final Behaviour Decomposition

Behaviour	Action Primitives	Sense Primitives
Player	connect idle	is_connected can_send_command is_alive
Game		can_move is_dead_ball is_our_dead_ball
Positional_Awareness		is_attacking is_my_nearest ball_location_known can_kick_ball is_too_close is_in_shooting_range is_near_own_goal is_pass_available is_at_home
Movement	go_home go_to_defensive_position stand_up go_to_attacking_position find_space face_ball run_to_ball turn	
Kicking	shoot clear pass_ball dribble	
Goalkeeper	go_to_gk_pos catch_ball	is_in_gk_pos _ can_catch_ball

In this final behaviour decomposition, the *player*, *movement* and *kicking* modules are used for the same reasons as in the initial decomposition, but are enriched with more primitives. Similarly, the *positional_awareness* behaviour module in the final decomposition serves the same purpose as the *vision* module in the initial decomposition, but was renamed to reflect that it also uses information received audibly and a player’s memory. The *game* behaviour is used to provide primitives relating to an agent’s awareness of the current game state, and the *goalkeeper* behaviour supports primitives which are applicable

only to a goalkeeper.

5.7.4 Strategy

Running in parallel to the development of new behaviours and improvement of plans for agents, I also adjusted by *init* file during each iteration of the development cycle. Although this *init* file exists purely to specify which agents should be created, this is of great significance in the multi agent system of a football team. This is because it is in this file where the roles of the individual players are defined. I tried to keep to a strategy whereby I would implement behaviours, and construct plans first, and then adjust *init* files to suit these. This meant that I could effectively segregate higher level team strategy as encompassed in the *init* file from individual player reasoning. Although they were segregated in this way, one still had an impact on the other during the next iteration of development, as the choice of the next area of behaviour to implement or adjust was based on analysis of team performance, which is affected by both of these areas.

5.8 Summary

This chapter has outlined the numerous concepts which were implemented in my football team, as well as providing some insight into the process which the development of this implementation followed. In the next chapter, I give an account of how the team performed.

Chapter 6

Results

6.1 Introduction

In this chapter, I describe how successful the team that I implemented was. I divide this into two main areas – against teams using entirely different behaviours, as produced by other researchers, and against teams using the same behaviours but different plans.

6.2 Performance Against other Available Teams

When I tested my team playing against teams which have made their source code freely available, my team suffered very heavy defeats in which my team was not competitive at all.

One reason that I have identified for this is the issue whereby actions are not always selected in time for the next action cycle, so in many action cycles, a player may take no action. This was as a result of one POSH cycle taking longer than a server cycle due to the computation involved in some of the implemented skills and percepts. This led to problems such as players executing a dash, then not executing a dash again in the next cycle where this would have been the selection made, which resulted in the player losing speed. This meant that my team suffered when playing teams which were better optimised in which actions were always selected for each action cycle for every player. This problem is made more obvious when competing against other teams, as it gives the impression that players on my team are generally slower at running than those on the opposing team. This means that opposition players are able to easily run past my team's defence.

Another reason for my team's poor performance in comparison with these other teams is as a result of my player's being aware of less of their surroundings than opposing players. The main reason for this is the fact that my players *look around* less. This is mainly as a result of the lack of use of the *turn neck* action which enables players to look in different directions while facing the

same direction with their body. I observed opposing teams using this action very regularly, apparently to great effect.

Along with the reasons already mentioned, a large reason for my team's poor performance against others is the difference in performance of low-level skills. This includes skills such as running to the ball, and shooting. As a result of my team's low-level skills being inferior to those of other teams, it means that assessing my team's strategies and plans by playing against these other teams is almost meaningless. This is because whatever plans my team were to employ, opposing teams would simply use their superior basic skills to overcome them with ease.

6.3 Performance Against Teams Using Same Behaviours

When running my team against other versions using the same behaviours, but with a different set of plans, I was able to assess the effectiveness of the plans which I created.

For example I set up a game between a team using my final plan against a team using a plan which varied only by meaning that players must try to remain in position at all times by heading to their home position when out of position. In this game, at certain times it was clear that players from the latter team failed to take full advantage of some situations as a result of this restriction on where players could go on the pitch, while the team using the final plan were free to take advantage of similar situations. The team using the final plan won this game 2-0.

Similarly to testing between teams using plans using the same sets of behaviours, I was also able to create games between teams using different subsets of the primitives made available by behaviours. For example, I set up a game between a team using my final plan against a team using a plan which varied only by not ever using the *stand_up* primitive (using the *go_to_defensive_position* primitive in its place). In this game, it could be seen how the team using the final plan was able to attack more effectively as a result of the opposing team not attempting to actively block players' paths. The team using the final plan won this game 2-0.

Through such testing against teams using the same set of behaviours, I was able to appreciate the effectiveness of some of the more complex strategies which were employed. It also highlighted the importance of well-constructed reactive plans in order to realise the potential of a set of behaviour modules.

6.4 Summary

In this chapter, I have described that although the behaviours implemented and other technical issues meant that the team was not competitive against some teams which had been made available, the ability to test between teams with

the same behaviour modules at their disposal gave positive results. In the next chapter, I discuss my findings both relating specifically to the simulated football team and as a result of my experience with BOD in general.

Chapter 7

Discussion

7.1 Introduction

In this chapter, I discuss some of the key areas of interest identified through carrying out the project. I firstly discuss areas which relate specifically to my SoccerServer team and then more generally to BOD.

7.2 Team Performance

The overall performance of the team was pleasing, with some reasonably advanced strategies being used, which were easily perceivable. Although the strategies being used were generally easily perceivable when the game was run at full speed, when two teams were being run on my machine at the same time, actions were not always sent to the server in time. This was as a result of the intensity of the level of computation involved for all 22 players. When the number of players running on my machine was reduced, agents always sent actions in time to the server. Similarly when the length of cycles was increased, actions were always sent to the server in time.

This could be improved by:

- Running agents on a more powerful machine
- Running agents on multiple, networked machines
- Optimising algorithms used by agents
- Simplifying agents

However, I was able to assess the performance of my team with the effects of this problem negated by playing the team against another team using my set of behaviours. This worked because both teams would be affected (roughly) equally by the problem.

When running the team in such “fair” games, it became clear how different areas of players’ plans combined with each other. Although it was possible to construct plans which would complement each other using some degree of logic, the highly dynamic nature of football meant that the results of some parts of plans being used together are very difficult to predict. It is for this reason that the best way to develop them is through viewing them in action. Through doing this, I managed to combine plans which worked together, often with positive complementary effects which I had not thought about before actually viewing them in action.

An example of this was as a result of the reactive plan used to decide when a player should pass the ball and the nature of a player’s memory model. Here, when a player who is the furthest up the pitch on their team runs to the ball in the direction of the opponents’ goal, they will disregard the choice of passing the ball to a teammate as a result of not seeing any for a while. This means that the player will choose to dribble the ball towards the goal which, as a result of the player being the most advanced on their team, will usually be the best option for maximising the team’s chances of scoring a goal.

Another example is as a result of a goalkeeper’s reactive plan which specifies that they should go to the ball when they are the nearest player on their team to it. The idea of this was to enable them to go to where a shot is headed in order to catch it, or to go to the ball when it is passed back to them. However, this aspect of a goalkeeper’s reactive plan had the unforeseen side-effect that when an opposing attacker makes a break past the defence, the goalkeeper will advance off their line, closing down the angle for a shot in a similar manner to that of human goalkeepers.

7.3 Behaviour-Oriented Design

7.3.1 Iterative Development

I found that the iterative nature of BOD was ideal for the design of a team for SoccerServer. This was because the dynamic nature of the domain meant that changes made to agents could have many unforeseen effects on areas of their performance other than those for which the change was intended to directly affect. Iterative design allowed me to analyse the unforeseen effects of such changes. I could then decide what should be implemented or adjusted next as a result of these effects. This allowed me to develop behaviours and plans in response to the effects of previous iterations, allowing for the positive evolution of created agents over iterations.

Without the benefits brought about by using such an iterative process, it would have been incredibly difficult to foresee *all* effects of any changes, and how they would interact with each other in such a dynamic environment. For this reason, using original intended designs may have had negative side effects on the performance of agents. The use of an iterative design process allows for using future design phases of development to address such issues.

7.3.2 Behaviour Decomposition

The way in which BOD encouraged me to think from a high level what the individual behaviour modules should be, and what action and sense primitives should be in these resulted in me producing code that maximised reusability. BOD encouraged these design decisions through the prescribing of behaviour decompositions. This was the case because it is through these decompositions that the developer is made to consider which areas of behaviour are distinct and how they should interact with each other.

This was beneficial because it meant that these actions and senses had to be implemented independently rather than procucing unweildy all-encompassing methods which cannot be easily altered or used elsewhere. This also maximised extensibility of my agents. This was a feature which I found very useful, as through analysis of my team’s performance, I frequently identified additional behaviours from which it would benefit. The modularity of my agents as a result of using BOD allowed me to easily slot in these additional behaviours without affecting those which were previously implemented.

7.3.3 Reactive Plans

The use of plans was very useful in enabling me to quickly and easily reuse behaviours with different triggers and priorities. This was very important, as it occurred many times that behaviours had been implemented satisfactorily, but I decided to change the situations in which they were used, which the use of POSH plans allowed me to do. It also allowed me to easily experiment with such plans in ways which I had no reason to suspect would improve overall performance of agents but due to the dynamic nature of the domain, did for unforeseen reasons. This scope for experimentation, coupled with BOD’s recommendation for keeping all plans meant that overall performance could be improved through experimentation, while keeping the option to revert to previous plans very accessible.

I found that in the creation of plans, although there were some instances in which the use of *action plans* initially seemed appropriate, the dynamic nature of the domain of simulated football generally meant that they were not a sensible choice. This is because the effectiveness of *any* action that is made in SoccerServer can in some way be affected by the innumerable posible changes occur to the world state outside of the player’s control. For this reason, there are *no* circumstances in which players should *always* follow one action with another. This meant that in later iterations of my team, action plans were not used at all. However, this is not a shortcoming of BOD, as the use of *competences* actually caters for these needs exactly, by allowing the use of triggers for certain actions to be executed.

7.3.4 Cycles

The manner in which SoccerServer works in cycles was problematic in terms of how I was able to use jyPOSH for my implementation. SoccerServer specifies the frequency of cycles in milliseconds, however it is not guaranteed that the exact specified frequency will be achieved. This caused the problem of configuring when the next *loop* should be fired in jyPOSH. This was because if SoccerServer were to run slower than expected and the plan was set for loops to be executed with a certain frequency, the agent may attempt to execute two or more actions in the same cycle. This would result in not all actions taking place, and also the agent's state would change under the assumption that the actions did take place, thus affecting future actions.

The workaround which I used to cope with this was to implement an extra *sense* primitive which would return whether it was legal for the agent to execute an action. This sense would then be used as a trigger for a *perform action* competence in the drive collection, with an *idle* action being performed in other cases. All of the actual decision making would then occur within the *perform action* competence. This workaround was very simple and effective.

7.3.5 Separation of Plans and Behaviours

The separation of the concepts of *plans* and *behaviours* was very useful in the project. This was because it allowed for me to easily use different plans with different players based on their role in the team, using the same set of underlying behaviours. However, I have identified that as the complexity of individual players increases, it may be useful for them to adopt different plans dynamically during the course of a game. For example, a point may be reached in the game where it is sensible to make an attacker become a defender in order to increase the team's defensive ability. To do this, this player must start using a *defender plan*, but jyPOSH does not provide a way to allow dynamic switching between plans, although such functionality can be obtained by using a more complex plan, where the *plans* which are to be switched between are implemented as competences within this *master* plan. Still, the functionality to be able to dynamically switch between plans as a feature of jyPOSH would be more intuitive, and be more self-documenting, as is an aim of BOD. This would also allow for a more modular approach to plans, which would contribute towards promoting the writing of reusable code. This is because multiple plans could be written, with different agents using different subsets of these plans in different situations.

7.3.6 Debugging

The debugging of plans could sometimes be a longer-than-expected process. This was the case when trying to check whether certain actions were triggered when expected, and was as a result of the difficulty of artificially creating particular game scenarios. For example, while the monitor provided with SoccerServer allows the user to award a free kick to either team, or drop the ball at any lo-

cation, there was no such functionality to award a throw in or corner. This meant that in testing whether plans worked as desired in such situations, either a large portion of a game had to be played, with the hope that such a scenario would occur, or players must be equipped with plans which would increase the likelihood of such events occurring. As a result of this, I found that I sometimes made a relatively large number of changes to behaviours and plans before testing, so going against a principle of BOD to test very frequently. I did this when the changes which I had made had no foreseeable impact on one another, and in running a game after making these sets of changes, I could test them all at once, so reducing the amount of time which had to be spent waiting for particular scenarios to occur.

However, this debugging process was as a result of the implementation of SoccerServer, and not that of jyPOSH. This did not, however, mean that debugging the action selection carried out by jyPOSH was straightforward apart from this. This is because when there were many players running at the same time, it could be difficult to find a way of tracking the action selection process of a specific player. There were ways of doing this, such as creating logs, and outputting action selection information along with player numbers, however these methods were not very effective for debugging plans while they were being used. This was because once I had interpreted the output information, the game scenario would have changed, and it would be very difficult for me to monitor a constant stream of such debugging information. For this reason, it may be beneficial to produce a more intuitive tool for debugging.

7.4 Summary

In this chapter I have discussed some of the issues which arose through the development of my SoccerServer team. I have also discussed my experience with certain aspects of BOD. In the next chapter, I develop the points of discussion to formulate possible future work to improve the team and enhance BOD.

Chapter 8

Future Work

8.1 Introduction

As a result of the work which I have carried out, I have identified a number of areas for which further work could be conducted.

The most immediate areas for future work which I have identified are largely outlined in the descriptions of skills and percepts. This is because with the improvement of low-level skills, the effectiveness of any plans using these skills can be improved. However, there are other areas which can also be worked on. I describe these in this chapter.

8.2 Percepts

A particular theme which emerged from identifying possible improvements to percepts is that of moving away from using percepts with boolean return values to ones which returned quantitative results. This would allow for more fine tuning of plans by tweaking triggers, and avoids having to adjust the implementations of percepts themselves, when only some threshold parameter needs to be changed. In this way, the same percepts could be used with different ranges of return values being used as triggers in different situations or for different players.

For example, the *is attacking* percept could be adjusted so that rather than giving a *true/false* return value, it could give the distance of the closest player on the player's team from the ball minus the distance of the closest opposing player from the ball. This would mean that, using the determining factor of being *in possession* to be the team to which the closest player to the ball belongs, a negative value would mean that the team was in possession and a positive value would mean they are not. Then, using this adjusted percept, rather than having players only run to the ball when they are the nearest player to the ball, the plan for an attacker could be reconfigured so that they would run to the ball if no opposing player were more than a distance x closer to the ball than them, while other players could use the percept as before. This would mean that the player

would try to win more contentious balls, while defenders would not attempt risky tackles.

8.3 SoccerServer Features

There are features made available by SoccerServer which I did not take advantage of. These are:

- The *turn neck* action
- The *change view* action
- The use of a *coach*

The overall performance of my team could be improved by exploring the use of these features.

In my implementation, I avoided the use of the *turn neck* action in order to keep things simple by allowing myself to use the assumption that a player's head would always be facing the same direction as their body. However, this restriction meant that when the ball was out of view and I wanted the player to look for the ball, this involved the player turning their body. This meant that the player could not execute a *dash* at the same time as turning to look for the ball. However, if I were to have used the *turn neck* functionality, the player could potentially have simply turned their neck to see the ball, and have continued to dash without losing any speed through turning. This would be particularly useful when players are moving to a tactical position which would involve running in a direction in which they could not see the ball if facing straight ahead, as they could keep looking at the ball while running into position.

8.4 Debugging

The issues which I had with debugging as outlined in Section 7.3.6, specifically relating to trying to test parts of plans which are used in specific scenarios, could be improved through the implementation of a tool to easily artificially create such scenarios.

Such a tool could work by allowing the developer to pause the plan execution of all players, then allowing for the state of the world to be manipulated. This could work by allowing the developer to specify desired positions for all players, and then actions would be executed by the players in order to get into these positions. When the world state was as desired, the tool could allow the developer to resume the execution of plans in order to test them in the world scenario that had been created.

Because it may be useful to test plans repeatedly in the same scenarios in order to get the plans right, the tool should be able to save world states that have been specified previously so that they can be used again. This would mean that the same scenarios would not have to be manually set up time and time again.

8.5 ABODE

The ABODE tool which currently exists is only partially implemented. It allows for POSH plans to be viewed in a graphical format. This is useful in visualising plans which have been constructed. However, as I implemented more complex POSH plans and it became more difficult to keep track of the structure of plans, it became clear that it would be useful to be able to develop the plans in a more visual manner. In order to allow for this, I started working on the existing ABODE tool to enable it to be used to easily develop plans. However, after some work on trying to adapt the tool from its current state, I identified fundamental problems with the existing partial implementation which meant that in order to do this effectively, a complete rehaul would be required.

In order to allow for POSH plans to be developed more easily, a new version of ABODE could be developed which allows for plans to be created and adjusted through an intuitive graphical interface. In addition to the tool enabling plan development, it could also have debugging features which would allow it to plug into jyPOSH and give visual feedback on which parts of a plan were being used when. This would be useful for identifying whether parts of plans were being triggered when they are designed to without having to use debugging methods within behaviours to work out which parts of plans are being executed.

8.6 Summary

Although I have identified how the implementation of the team can be directly improved, it is the work on tools to aid this improvement which should be prioritised. This is because such tools would potentially accelerate this process. In particular, work on ABODE would be of particular value because as well as this allowing for more complex plans to be more easily implemented for a simulated football team, this would also be applicable to other projects in which BOD is used.

Chapter 9

Conclusion

In creating my team for SoccerServer, I have gained first-hand experience of using BOD in a domain for which it had not previously been applied. My experience with the methodology was very positive and I felt that it was well-suited to the task in hand. Although I experienced a number of difficulties in my implementation of a simulated football team, these were generally as a result of the problem domain, rather than as a result of using BOD. More than anything else, this highlighted how the potential of the use of advanced reactive plans can only be truly realised in conjunction with effective behaviours.

I also identified areas for which the tools used for carrying out BOD could be enhanced in small ways, which would make projects such as this easier or more intuitive to carry out. These did not entail any fundamental changes to the methodology.

Bibliography

- Abdelaziz Tawfig, M. (2008). Towards a comprehensive agent-oriented software engineering methodology.
- Amiri, F., Bashiri, H., Kafi, S., Kaviani, N., Rafaie, M., and Zarrabi-Zadeh, H. (2004). Robosina 2004 team description.
- Bowling, M., Stone, P., and Veloso, M. (1996). Predictive memory for an inaccessible environment. In *Working Notes of the IROS-96 Workshop on RoboCup*.
- Bresciani, P., Giorgini, P., Giunchiglia, F., Mylopoulos, J., and Perini, A. (2003). Tropos: An agent-oriented software development methodology.
- Bryson, J. J. (2003). The behavior-oriented design of modular agent intelligence. In *Agent Technologies, Infrastructures, Tools, and Applications for e-Services*, pages 61–76. Springer.
- Cockburn, A. (2000). Selecting a project’s methodology. *IEEE Software*, 17(4):64–71.
- Dam, K. H. (2003). Comparing agent-oriented methodologies. pages 78–93.
- de Boer, R., Kok, J., and Groen, F. (2001). UvA Trilearn 2001 team description. In *RoboCup-2001: Robot Soccer World Cup V*. Springer Verlag.
- DeLoach, S. A., Wood, M. F., and Sparkman, C. H. (2001). Multiagent systems engineering.
- Gabel, T. and Veloso, M. (2001). Selecting heterogeneous team players by case-based reasoning: A case study in robotic soccer simulation.
- Garcia, E., Giret, A., and Botti, V. (2011). Evaluating software engineering techniques for developing complex systems with multiagent approaches. *Information and Software Technology*, 53(5):494 – 506. Special Section on Best Papers from XP2010.
- Igarashi, K. S. H. (1999). RoboCup-99 Simulation League: Team KU-Sakura2.
- Kitano, H., Asada, M., Noda, I., and Matsubara, H. (1998). RoboCup: robot world cup. *IEEE Robotics & Automation Magazine*, 5(3):30–36.

- Kose, H., Celik, B., and Akin, H. L. (2008). Comparison of localization methods for a robot soccer team. *International Journal of Advanced Robotic Systems*, 3(4).
- Luck, M., McBurney, P., and Gonzalez-Palacios, J. (2006). Agent-based computing and programming of agent systems. In *Programming Multi-Agent Systems, volume 3862 of Lecture*, pages 23–37. Springer.
- Luck, M., McBurney, P., Shehory, O., and Willmott, S. (2005). *Agent Technology: Computing as Interaction (A Roadmap for Agent Based Computing)*. AgentLink.
- Nakagawa, K., Asai, N., Ito, N., Du, X., and Ishii, N. (2000). NITStones-99. In *RoboCup-99: Robot Soccer World Cup III*, pages 608–610, London, UK. Springer-Verlag.
- Noda, I., Matsubara, H., Hiraki, K., and Frank, I. (1997). Soccer Server: a tool for research on multi-agent systems. *Applied Artificial Intelligence*, 12:233–250.
- Padgham, L. (2005). Tool support for agent development using the Prometheus methodology. In *First international workshop on Integration of Software Engineering and Agent Technology (ISEAT 2005)*.
- Padgham, L. and Winikoff, M. (2009). Prometheus: A methodology for developing intelligent agents.
- Penedo, C., Pavo, J., Lima, P., and Ribeiro, M. I. (2003). Markov Localization in the RoboCup Simulation League 1.
- Polani, D., Weber, S., and Uthmann, T. (1998). The Mainz Rolling Brains in RoboCup ’98: A direct approach to robot soccer.
- Reis, L. P. and Lau, N. (2001). FC Portugal team description: RoboCup 2000 Simulation League Champion. In *RoboCup-2000: Robot Soccer World Cup IV*, pages 29–40. Springer Verlag.
- Stone, P. and Veloso, M. (1998). The CMUnited-97 simulator team.
- Stone, P. and Veloso, M. (1999). Task decomposition, dynamic role assignment, and low-bandwidth communication for real-time strategic teamwork. *Artificial Intelligence*, 110(2):241–273.
- Stone, P., Veloso, M., and Riley, P. (1999). The CMUnited-98 champion simulator team. In *RoboCup-99: Robot Soccer World Cup III*, pages 35–48. Springer Verlag.
- Wang, C., Chen, X., Zhao, X., and Ju, S. (2008). Design and implementation of a general decision-making model in RoboCup simulation. *International Journal of Advanced Robotic Systems*, 1(3).

- Wooldridge, M., Jennings, N. R., and Kinny, D. (1999). A methodology for agent-oriented analysis and design.
- Yang, G., Junfeng, L., Lihui, C., and Feng, P. (2002). Everest 2002 team description. In *Proceedings of RoboCup-2002: Robot Soccer World Cup VI*.
- Yusof, M. M., Shukur, Z., and Abdullah, A. L. (2011). CuQuP: A hybrid approach for selecting suitable information systems development methodology. *Information Technology Journal*, 10(5):1031–1037.

Appendix A

Plan Files

During the implementation of my football team, I wrote and edited a large number of plan files. Some of these were attempts to create the best team possible, some were written to test specific parts of large plans for which they were to form a part of, and some were written to get agents to exhibit a certain behaviour so as to test how *other* agents would react to this.

Here are the two plans which I found to be most successful – one for a goalkeeper, and the other for outfield players:

Listing A.1: player.lap

```
1  (
2    (C play
3      (elements
4        (
5          (do-go-home (trigger ((can-move 1))) go-home)
6          (do-attack (trigger ((is-attacking 1))) attack)
7          (do-defend (trigger ((is-attacking 0))) defend)
8        )
9      )
10   )
11
12   (C attack
13     (elements
14       (
15         (do-nearest-attack (trigger ((is-my-nearest 1)))
16                               nearest-attack)
17         (do-not-nearest-attack (trigger ((is-my-nearest 0)))
18                                   not-nearest-attack)
19       )
20     )
21   )
22   (C nearest-attack
23     (elements
```

```

23      (
24      (do-kick (trigger ((can-kick-ball 1))) kick-ball)
25      (find-ball (trigger ((ball-location-known 0))) turn)
26      (run-to-ball (trigger ((ball-location-known 1)))
        run_to_ball)
27      )
28  )
29  )
30
31  (C not-nearest-attack
32  (elements
33  (
34      (do-kick (trigger ((can-kick-ball 1))) kick-ball)
35      (find-ball (trigger ((ball-location-known 0))) turn)
36      (get-in-position (trigger ((ball-location-known 1)))
        go_to_attacking_position)
37  )
38  )
39  )
40
41  (C defend
42  (elements
43  (
44      (do-nearest-defend (trigger ((is-my-nearest 1)))
        nearest-defend)
45      (do-not-nearest-defend (trigger ((is-my-nearest 0)))
        not-nearest-defend)
46  )
47  )
48  )
49
50  (C nearest-defend
51  (elements
52  (
53      (do-kick (trigger ((can-kick-ball 1))) kick-ball)
54      (find-ball (trigger ((ball-location-known 0))) turn)
55      (go-to-dead-ball (trigger ((is-our-dead-ball 1)))
        run_to_ball)
56      (stand-up (trigger ((is-my-nearest 1))) stand-up)
57  )
58  )
59  )
60
61  (C not-nearest-defend
62  (elements
63  (
64      (do-kick (trigger ((can-kick-ball 1))) kick-ball)
65      (find-ball (trigger ((ball-location-known 0))) turn)
66      (go-defend (trigger ((is-my-nearest 0)))
        go_to_defensive_position)

```

```

67     )
68   )
69 )
70
71 (C kick-ball
72   (elements
73     (
74       (do-shoot (trigger ((is_in_shooting_range 1))) shoot)
75       (do-clear (trigger ((is_near_own_goal 1))) shoot)
76       (do-pass (trigger ((is_pass_available 1))) pass_ball)
77       (do-dribble (trigger ((is_pass_available 0) (
78         is_dead_ball 0))) dribble)
79       (do-boot (trigger ((is_pass_available 0))) shoot)
80     )
81   )
82
83 (SDC life
84   (drives
85     (
86       (do-connect (trigger ((is_connected 0))) connect)
87       (perform-action (trigger ((can_send_command 1))) play)
88       (wait (trigger ((is_alive 1))) idle)
89     )
90   )
91 )
92 )

```

Listing A.2: goalkeeper.lap

```

1  (
2    (C cycle-command
3      (elements
4        (
5          (go-home (trigger ((can_move 1))) go_home)
6          (do-shoot (trigger ((can_kick_ball 1))) shoot)
7          (do-catch (trigger ((can_catch_ball 1))) catch_ball)
8          (find-ball (trigger ((ball_location_known 0))) turn)
9          (do-run (trigger ((ball_location_known 1) (
10             is_my_nearest 1))) run_to_ball)
11          (get-in-position (trigger ((is_in_gk_pos 0)))
12            go_to_gk_pos)
13          (face-ball (trigger ((is_alive 1))) face_ball)
14        )
15      )
16    )
17    (SDC life
18      (drives

```

```
19      (  
20      (do-connect (trigger ((is-connected 0))) connect)  
21      (perform-action (trigger ((can-send-command 1))) cycle  
      -command)  
22      (wait (trigger ((is-alive 1))) idle)  
23      )  
24  )  
25 )  
26 )
```

Appendix B

Code Listings

Here is some of the code used in my SoccerServer team (note, plans are not included here – they are included in Appendix A).

First, I list the *Behaviour Modules*, written in jython, then a selection of *init_agent* files that can be used for different configurations of teams and finally some classes which form a part of the Java code which provides the skills and percepts used by agents and interfacing with SoccerServer. The code not included in this appendix can be found on the enclosed CD.

In the Java code, the Client class is what the Player behaviour module directly interfaces with and the RCSSServer class provides an interface with SoccerServer. The Parser class is used to parse communications received from SoccerServer. Much of a player's decision-making is dealt with by Controlled-Player.

B.1 Behaviour Modules

Listing B.1: player.py

```
1 from __future__ import nested_scopes
2 from POSH.jython.compat import *
3
4 from POSH import Behaviour
5
6 from behaviours import Client
7 from world import Coordinate, Rectangle
8
9 class Player(Behaviour):
10
11     def __init__(self, agent):
12         Behaviour.__init__(self, agent,
13                             ("connect", "idle", ),
14                             ("is_connected", "can_send_command",
15                              "is_alive", ))
16
17         self.team_name = "tomsteam"
18         # The player's home coordinates
19         self.x = -20
20         self.y = 0
21
22         # The player's rectangle
23         self.x1 = -57
24         self.x2 = 57
25         self.y1 = -36
26         self.y2 = 36
27
28         self.goalkeeper = False
29
30         self.javaClient = None
31
32     def check_error(self):
33         return False
34
35     def exit_prepare(self):
36         pass
37
38     # Percepts
39
40     def is_connected(self):
41         return self.javaClient != None
42
43     def can_send_command(self):
44         return self.is_connected() and self.javaClient.
45             canSendCommand()
46
47     def is_alive(self):
48         return True
49
50     # Actions
```

```
49
50     def connect(self):
51         self.javaClient = Client(self.team_name, Coordinate(
52             self.x, self.y),
53                                     Rectangle(self.x1, self.x2,
54                                                self.y1, self.y2),
55                                     self.goalkeeper)
56
57         return True
58
59     def shoot(self):
60         self.javaClient.shoot()
61         return True
62
63     def clear(self):
64         self.javaClient.shoot()
65         return True
66
67     def pass_ball(self):
68         self.javaClient.passBall()
69         return True
70
71     def dribble(self):
72         self.javaClient.dribble()
73         return True
74
75     def kick(self):
76         self.javaClient.kick(100, 0)
77         return True
78
79     def idle(self):
80         return True
```

Listing B.2: game.py

```
1 from __future__ import nested_scopes
2 from POSH.jython.compat import *
3
4 from POSH import Behaviour
5
6 class Game(Behaviour):
7
8
9     def __init__(self, agent):
10         Behaviour.__init__(self, agent,
11                             (),
12                             ("can_move", "is_dead_ball", "
13                              is_our_dead_ball", ))
14
15     def check_error(self):
16         return False
17
18     def exit_prepare(self):
19         pass
20
21     # Percepts
```



```

21
22 def can_move(self):
23     return self.agent.Player.javaClient.canMove()
24
25 def is_dead_ball(self):
26     return self.agent.Player.javaClient.isDeadBall()
27
28 def is_our_dead_ball(self):
29     return self.agent.Player.javaClient.isOurDeadBall()

```

Listing B.3: positional_awareness.py

```

1 from __future__ import nested_scopes
2 from POSH.jython_compat import *
3
4 from POSH import Behaviour
5
6 class PositionalAwareness(Behaviour):
7
8
9     def __init__(self, agent):
10         Behaviour.__init__(self, agent,
11                             (),
12                             ("is_attacking", "is_my_nearest", "
                                ball_location_known", "
                                can_kick_ball", "is_too_close",
                                "is_in_shooting_range", "
                                is_near_own_goal", "
                                is_pass_available", "
                                is_in_position", "is_at_home",
                                ))
13
14     def check_error(self):
15         return False
16
17     def exit_prepare(self):
18         pass
19
20     # Percepts
21
22     def can_move(self):
23         return self.agent.Player.javaClient.canMove()
24
25     def ball_location_known(self):
26         return self.agent.Player.javaClient.
27             isBallLocationKnown()
28
29     def can_kick_ball(self):
30         return self.agent.Player.javaClient.canKickBall()
31
32     def is_attacking(self):
33         return self.agent.Player.javaClient.isAttacking()
34
35     def is_my_nearest(self):
36         return self.agent.Player.javaClient.isMyNearest()

```

```

36
37 def is_too_close(self):
38     return self.agent.Player.javaClient.isTooClose()
39
40 def is_in_shooting_range(self):
41     return self.agent.Player.javaClient.isInShootingRange()
42
43 def is_near_own_goal(self):
44     return self.agent.Player.javaClient.isNearOwnGoal()
45
46 def is_pass_available(self):
47     return self.agent.Player.javaClient.isPassAvailable()
48
49 def is_in_position(self):
50     return self.agent.Player.javaClient.isInPosition()
51
52 def is_at_home(self):
53     return self.agent.Player.javaClient.isAtHome()

```

Listing B.4: movement.py

```

1 from __future__ import nested_scopes
2 from POSH.jython_compat import *
3
4 from POSH import Behaviour
5
6 class Movement(Behaviour):
7
8
9     def __init__(self, agent):
10         Behaviour.__init__(self, agent,
11                             ("go_home", "
                                go_to_defensive_position", "
                                stand_up", "
                                go_to_attacking_position", "
                                find_space", "face_ball", "
                                run_to_ball", "turn", ),
12                             ( ))
13
14         self.javaClient = None
15
16     def check_error(self):
17         return False
18
19     def exit_prepare(self):
20         pass
21
22     # Actions
23
24     def go_home(self):
25         self.agent.Player.javaClient.goHome()
26         return True
27
28     def go_to_defensive_position(self):

```

```

29         self.agent.Player.javaClient.goToDefensivePosition()
30         return True
31
32     def go_to_attacking_position(self):
33         self.agent.Player.javaClient.goToAttackingPosition()
34         return True
35
36     def stand_up(self):
37         self.agent.Player.javaClient.standUp()
38         return True
39
40     def find_space(self):
41         self.agent.Player.javaClient.findSpace()
42         return True
43
44     def face_ball(self):
45         self.agent.Player.javaClient.faceBall()
46         return True
47
48     def run_to_ball(self):
49         self.agent.Player.javaClient.runToBall(100)
50         return True
51
52     def turn(self):
53         self.agent.Player.javaClient.turn(45)
54         return True

```

Listing B.5: kicking.py

```

1 from __future__ import nested_scopes
2 from POSH.jython_compat import *
3
4 from POSH import Behaviour
5
6 class Kicking(Behaviour):
7
8
9     def __init__(self, agent):
10         Behaviour.__init__(self, agent,
11                             ("shoot", "clear", "pass_ball", "
12                                dribble", ),
13                             ())
14
15     def check_error(self):
16         return False
17
18     def exit_prepare(self):
19         pass
20
21     # Actions
22
23     def shoot(self):
24         self.agent.Player.javaClient.shoot()
25         return True

```

```

26 def clear(self):
27     self.agent.Player.javaClient.shoot()
28     return True
29
30 def pass_ball(self):
31     self.agent.Player.javaClient.passBall()
32     return True
33
34 def dribble(self):
35     self.agent.Player.javaClient.dribble()
36     return True

```

Listing B.6: goalkeeper.py

```

1 from __future__ import nested_scopes
2 from POSH.jython_compat import *
3
4 from POSH import Behaviour
5
6 class Goalkeeper(Behaviour):
7
8
9     def __init__(self, agent):
10         Behaviour.__init__(self, agent,
11                             ("go_to_gk_pos", "catch_ball", ),
12                             ("can_catch_ball", "is_in_gk_pos",
13                                ))
14
15         self.team_name = "tomsteam"
16
17     def check_error(self):
18         return False
19
20     def exit_prepare(self):
21         pass
22
23     # Percepts
24
25     def can_catch_ball(self):
26         return self.agent.Player.javaClient.canCatchBall()
27
28     def is_in_gk_pos(self):
29         return self.agent.Player.javaClient.isInGKPos()
30
31     # Actions
32
33     def go_to_gk_pos(self):
34         self.agent.Player.javaClient.goToGKPos()
35         return True
36
37     def catch_ball(self):
38         self.agent.Player.javaClient.catchBall()
39         return True

```

B.2 Init Files

Listing B.7: One Team – 4-4-2

1 ### TEAM 1 ###
2
3 # GK
4 [goalkeeper]
5 Player.x = -50
6 Player.y = 0
7 Player.team.name = team.a
8 Player.x1 = -57
9 Player.x2 = -45
10 Player.y1 = -13
11 Player.y2 = 13
12 Player.goalkeeper = True
13
14 # RB
15 [player]
16 Player.x = -40
17 Player.y = -21
18 Player.team.name = team.a
19 Player.x1 = -57
20 Player.x2 = -10
21 Player.y1 = -39
22 Player.y2 = 0
23
24 # RCB
25 [player]
26 Player.x = -40
27 Player.y = -7
28 Player.team.name = team.a
29 Player.x1 = -57
30 Player.x2 = -10
31 Player.y1 = -20
32 Player.y2 = 10
33
34 # LCB
35 [player]
36 Player.x = -40
37 Player.y = 7
38 Player.team.name = team.a
39 Player.x1 = -57
40 Player.x2 = -10
41 Player.y1 = -10
42 Player.y2 = 20
43
44 # LB
45 [player]
46 Player.x = -40
47 Player.y = 21
48 Player.team.name = team.a
49 Player.x1 = -57
50 Player.x2 = -10

51 Player.y1 = 0
52 Player.y2 = 39
53
54 # RM
55 [player]
56 Player.x = -20
57 Player.y = -21
58 Player.team.name = team.a
59 Player.x1 = -52
60 Player.x2 = 57
61 Player.y1 = -39
62 Player.y2 = 0
63
64 # RCM
65 [player]
66 Player.x = -20
67 Player.y = -7
68 Player.team.name = team.a
69 Player.x1 = -40
70 Player.x2 = 36
71 Player.y1 = -25
72 Player.y2 = 15
73
74 # LCM
75 [player]
76 Player.x = -20
77 Player.y = 7
78 Player.team.name = team.a
79 Player.x1 = -40
80 Player.x2 = 36
81 Player.y1 = -15
82 Player.y2 = 25
83
84 # LM
85 [player]
86 Player.x = -20
87 Player.y = 21
88 Player.team.name = team.a
89 Player.x1 = -52
90 Player.x2 = 57
91 Player.y1 = 0
92 Player.y2 = 39
93
94 # RCF
95 [player]
96 Player.x = -5
97 Player.y = -2
98 Player.team.name = team.a
99 Player.x1 = -10
100 Player.x2 = 57
101 Player.y1 = -30
102 Player.y2 = 10
103
104 # LCF

```

105 [player]
106 Player.x = -5
107 Player.y = 2
108 Player.team_name = team.a
109 Player.x1 = -40
110 Player.x2 = 36
111 Player.y1 = -10
112 Player.y2 = 30

```

Listing B.8: Two Teams – 4-4-2

68

```

1  ### TEAM 1 ###
2
3  # GK
4  [goalkeeper]
5  Player.x = -50
6  Player.y = 0
7  Player.team_name = team.a
8  Player.x1 = -57
9  Player.x2 = -45
10 Player.y1 = -13
11 Player.y2 = 13
12 Player.goalkeeper = True
13
14 # RB
15 [player]
16 Player.x = -40
17 Player.y = -21
18 Player.team_name = team.a
19 Player.x1 = -57
20 Player.x2 = -10
21 Player.y1 = -39
22 Player.y2 = 0
23
24 # RCB
25 [player]
26 Player.x = -40
27 Player.y = -7
28 Player.team_name = team.a
29 Player.x1 = -57
30 Player.x2 = -10
31 Player.y1 = -20
32 Player.y2 = 10
33
34 # LCB
35 [player]
36 Player.x = -40
37 Player.y = 7
38 Player.team_name = team.a
39 Player.x1 = -57
40 Player.x2 = -10
41 Player.y1 = -10
42 Player.y2 = 20
43
44 # LB

```

```

45 [player]
46 Player.x = -40
47 Player.y = 21
48 Player.team_name = team.a
49 Player.x1 = -57
50 Player.x2 = -10
51 Player.y1 = 0
52 Player.y2 = 39
53
54 # RM
55 [player]
56 Player.x = -20
57 Player.y = -21
58 Player.team_name = team.a
59 Player.x1 = -52
60 Player.x2 = 57
61 Player.y1 = -39
62 Player.y2 = 0
63
64 # RCM
65 [player]
66 Player.x = -20
67 Player.y = -7
68 Player.team_name = team.a
69 Player.x1 = -40
70 Player.x2 = 36
71 Player.y1 = -25
72 Player.y2 = 15
73
74 # LCM
75 [player]
76 Player.x = -20
77 Player.y = 7
78 Player.team_name = team.a
79 Player.x1 = -40
80 Player.x2 = 36
81 Player.y1 = -15
82 Player.y2 = 25
83
84 # LM
85 [player]
86 Player.x = -20
87 Player.y = 21
88 Player.team_name = team.a
89 Player.x1 = -52
90 Player.x2 = 57
91 Player.y1 = 0
92 Player.y2 = 39
93
94 # RCF
95 [player]
96 Player.x = -5
97 Player.y = -2
98 Player.team_name = team.a

```

```

99 Player.x1 = -10
100 Player.x2 = 57
101 Player.y1 = -30
102 Player.y2 = 10
103
104 # LCF
105 [player]
106 Player.x = -5
107 Player.y = 2
108 Player.team_name = team.a
109 Player.x1 = -40
110 Player.x2 = 36
111 Player.y1 = -10
112 Player.y2 = 30
113
114
115 ### TEAM 2 ###
116
117 # GK
118 [goalkeeper]
119 Player.x = -50
120 Player.y = 0
121 Player.team_name = team.b
122 Player.x1 = -57
123 Player.x2 = -45
124 Player.y1 = -13
125 Player.y2 = 13
126 Player.goalkeeper = True
127
128 # RB
129 [player]
130 Player.x = -40
131 Player.y = -21
132 Player.team_name = team.b
133 Player.x1 = -57
134 Player.x2 = -10
135 Player.y1 = -39
136 Player.y2 = 0
137
138 # RCB
139 [player]
140 Player.x = -40
141 Player.y = -7
142 Player.team_name = team.b
143 Player.x1 = -57
144 Player.x2 = -10
145 Player.y1 = -20
146 Player.y2 = 10
147
148 # LCB
149 [player]
150 Player.x = -40
151 Player.y = 7
152 Player.team_name = team.b

```

```

153 Player.x1 = -57
154 Player.x2 = -10
155 Player.y1 = -10
156 Player.y2 = 20
157
158 # LB
159 [player]
160 Player.x = -40
161 Player.y = 21
162 Player.team_name = team.b
163 Player.x1 = -57
164 Player.x2 = -10
165 Player.y1 = 0
166 Player.y2 = 39
167
168 # RM
169 [player]
170 Player.x = -20
171 Player.y = -21
172 Player.team_name = team.b
173 Player.x1 = -52
174 Player.x2 = 57
175 Player.y1 = -39
176 Player.y2 = 0
177
178 # RCM
179 [player]
180 Player.x = -20
181 Player.y = -7
182 Player.team_name = team.b
183 Player.x1 = -40
184 Player.x2 = 36
185 Player.y1 = -25
186 Player.y2 = 15
187
188 # LCM
189 [player]
190 Player.x = -20
191 Player.y = 7
192 Player.team_name = team.b
193 Player.x1 = -40
194 Player.x2 = 36
195 Player.y1 = -15
196 Player.y2 = 25
197
198 # LM
199 [player]
200 Player.x = -20
201 Player.y = 21
202 Player.team_name = team.b
203 Player.x1 = -52
204 Player.x2 = 57
205 Player.y1 = 0
206 Player.y2 = 39

```

```

207
208 # RCF
209 [player]
210 Player.x = -5
211 Player.y = -2
212 Player.team_name = team.b
213 Player.x1 = -10
214 Player.x2 = 57
215 Player.y1 = -30
216 Player.y2 = 10
217
218 # LCF
219 [player]
220 Player.x = -5
221 Player.y = 2
222 Player.team_name = team.b
223 Player.x1 = -40
224 Player.x2 = 36
225 Player.y1 = -10
226 Player.y2 = 30

```

Listing B.9: Two Teams – 4-4-2 v 4-3-3

91

```

1  ### TEAM 1 ###
2
3  # GK
4  [goalkeeper]
5  Player.x = -50
6  Player.y = 0
7  Player.team_name = team.a
8  Player.x1 = -57
9  Player.x2 = -45
10 Player.y1 = -13
11 Player.y2 = 13
12 Player.goalkeeper = True
13
14 # RB
15 [player]
16 Player.x = -40
17 Player.y = -21
18 Player.team_name = team.a
19 Player.x1 = -57
20 Player.x2 = -10
21 Player.y1 = -39
22 Player.y2 = 0
23
24 # RCB
25 [player]
26 Player.x = -40
27 Player.y = -7
28 Player.team_name = team.a
29 Player.x1 = -57
30 Player.x2 = -10
31 Player.y1 = -20
32 Player.y2 = 10

```

```

33
34 # LCB
35 [player]
36 Player.x = -40
37 Player.y = 7
38 Player.team_name = team.a
39 Player.x1 = -57
40 Player.x2 = -10
41 Player.y1 = -10
42 Player.y2 = 20
43
44 # LB
45 [player]
46 Player.x = -40
47 Player.y = 21
48 Player.team_name = team.a
49 Player.x1 = -57
50 Player.x2 = -10
51 Player.y1 = 0
52 Player.y2 = 39
53
54 # RM
55 [player]
56 Player.x = -20
57 Player.y = -21
58 Player.team_name = team.a
59 Player.x1 = -52
60 Player.x2 = 57
61 Player.y1 = -39
62 Player.y2 = 0
63
64 # RCM
65 [player]
66 Player.x = -20
67 Player.y = -7
68 Player.team_name = team.a
69 Player.x1 = -40
70 Player.x2 = 36
71 Player.y1 = -25
72 Player.y2 = 15
73
74 # LCM
75 [player]
76 Player.x = -20
77 Player.y = 7
78 Player.team_name = team.a
79 Player.x1 = -40
80 Player.x2 = 36
81 Player.y1 = -15
82 Player.y2 = 25
83
84 # LM
85 [player]
86 Player.x = -20

```

```

87 Player.y = 21
88 Player.team_name = team_a
89 Player.x1 = -52
90 Player.x2 = 57
91 Player.y1 = 0
92 Player.y2 = 39
93
94 # RCF
95 [player]
96 Player.x = -5
97 Player.y = -2
98 Player.team_name = team_a
99 Player.x1 = -10
100 Player.x2 = 57
101 Player.y1 = -30
102 Player.y2 = 10
103
104 # LCF
105 [player]
106 Player.x = -5
107 Player.y = 2
108 Player.team_name = team_a
109 Player.x1 = -40
110 Player.x2 = 36
111 Player.y1 = -10
112 Player.y2 = 30
113
114
115
116 ### TEAM 2 ###
117
118 # GK
119 [goalkeeper]
120 Player.x = -50
121 Player.y = 0
122 Player.team_name = team_b
123 Player.x1 = -57
124 Player.x2 = -45
125 Player.y1 = -13
126 Player.y2 = 13
127 Player.goalkeeper = True
128
129 # RB
130 [player]
131 Player.x = -40
132 Player.y = -21
133 Player.team_name = team_b
134 Player.x1 = -57
135 Player.x2 = -10
136 Player.y1 = -39
137 Player.y2 = 0
138
139 # RCB
140 [player]

```

```

141 Player.x = -40
142 Player.y = -7
143 Player.team_name = team_b
144 Player.x1 = -57
145 Player.x2 = -10
146 Player.y1 = -20
147 Player.y2 = 10
148
149 # LCB
150 [player]
151 Player.x = -40
152 Player.y = 7
153 Player.team_name = team_b
154 Player.x1 = -57
155 Player.x2 = -10
156 Player.y1 = -10
157 Player.y2 = 20
158
159 # LB
160 [player]
161 Player.x = -40
162 Player.y = 21
163 Player.team_name = team_b
164 Player.x1 = -57
165 Player.x2 = -10
166 Player.y1 = 0
167 Player.y2 = 39
168
169 # RCM
170 [player]
171 Player.x = -20
172 Player.y = -15
173 Player.team_name = team_b
174 Player.x1 = -40
175 Player.x2 = 36
176 Player.y1 = -39
177 Player.y2 = 0
178
179 # CM
180 [player]
181 Player.x = -20
182 Player.y = 0
183 Player.team_name = team_b
184 Player.x1 = -40
185 Player.x2 = 36
186 Player.y1 = -15
187 Player.y2 = 15
188
189 # LCM
190 [player]
191 Player.x = -20
192 Player.y = 15
193 Player.team_name = team_b
194 Player.x1 = -40

```

```

195 Player.x2 = 36
196 Player.y1 = 0
197 Player.y2 = 39
198
199 # RF
200 [player]
201 Player.x = -5
202 Player.y = -15
203 Player.team_name = team.b
204 Player.x1 = -10
205 Player.x2 = 57
206 Player.y1 = 0
207 Player.y2 = -39
208
209 # CF
210 [player]
211 Player.x = -5
212 Player.y = -2
213 Player.team_name = team.b
214 Player.x1 = -10
215 Player.x2 = 57
216 Player.y1 = -30
217 Player.y2 = 10
218
219 # LF
220 [player]
221 Player.x = -5
222 Player.y = 15
223 Player.team_name = team.b
224 Player.x1 = -10
225 Player.x2 = 57
226 Player.y1 = 0
227 Player.y2 = 39

```

B.3 Java Code

Listing B.10: Client.java

```

1 package behaviours;
2
3 import java.util.HashMap;
4 import java.util.Random;
5
6 import world.*;
7 import world.GameObject.Side;
8
9 public class Client {
10
11     private String      teamName      = "TeamTom";
12     private String      version       = "15.0";
13     private RCSSServer   rcssServer    = null;
14     private ControlledPlayer controlledPlayer = new
        ControlledPlayer(this);
15     private String      gameState     = null;

```

```

16     private HashMap<String, String> serverParams    = new
        HashMap<String, String>();
17     private long          lastSentCommand    = 0;
18     private int           lastSeeCycle       = 0;
19     private int           cycle              = 0; // The current game
        cycle
20     private BodyMonitor   bodyMonitor        = new BodyMonitor
        ();
21     private boolean       ready              = true;
22
23     /**
24      * For debugging purposes
25      * @param args
26      */
27     public static void main(String[] args) {
28     }
29
30     /**
31      * Constructor with default variables
32      */
33     public Client() {
34         rcssServer = new RCSSServer(this);
35     }
36
37     public Client(String name) {
38         teamName = name;
39         rcssServer = new RCSSServer(this);
40     }
41
42     public Client(String name, Coordinate position) {
43         teamName = name;
44         controlledPlayer.setTeamPosition(position);
45         rcssServer = new RCSSServer(this);
46     }
47
48     public Client(String name, Coordinate position, Rectangle
        rect, boolean goalkeeper) {
49         teamName = name;
50         controlledPlayer.setTeamPosition(position);
51         controlledPlayer.setRectangle(rect);
52         controlledPlayer.setGoalkeeper(goalkeeper);
53         rcssServer = new RCSSServer(this);
54     }
55     public String getTeamName() {
56         return teamName;
57     }
58
59     public String getVersion() {
60         return version;
61     }
62
63     public ControlledPlayer getControlledPlayer() {
64         return controlledPlayer;
65     }

```



```

66
67 public String getGameState() {
68     return gameState;
69 }
70
71 public void setGameState(String gameState) {
72     if (gameState.equals("kick-off.l") || gameState.equals("
73         kick-off.r")) {
74         controlledPlayer.getWorld().getBall().setPosition(new
75             Coordinate(0,0));
76         controlledPlayer.getWorld().getBall().setVelocity(new
77             Coordinate(0,0));
78         controlledPlayer.getWorld().getBall().setSeen(cycle);
79     }
80     this.gameState = gameState;
81 }
82
83 public HashMap<String, String> getServerParams() {
84     return serverParams;
85 }
86
87 public void setServerParams(HashMap<String, String>
88     serverParams) {
89     this.serverParams = serverParams;
90 }
91
92 public void setCycle(int cycle) {
93     this.cycle = cycle;
94 }
95
96 public BodyMonitor getBodyMonitor() {
97     return bodyMonitor;
98 }
99
100 // Sets the number of the controlled player
101 public void setNumber(int number) {
102     controlledPlayer.setNumber(number);
103 }
104
105 /**
106  * Updates the lastSentCommand to the current time
107  */
108 public void updateLastSentCommand() {
109     this.lastSentCommand = System.currentTimeMillis();
110 }
111
112 /**
113  * Deals with any messages received from RCSSServer. (
114  * invoked from RCSSServer class)
115  * @param received The message received from the RCSSServer
116  */
117 public void react(String received) {
118     if (rcssServer != null) {
119         Parser parser = new Parser(this);

```

```

115     parser.Parse(this, received);
116 }
117 }
118
119 /**
120  * Sets the player controlled by this client to be on the
121  * given side
122  * @param side char (l or r) representing the side the
123  * player controlled
124  * by this client is on
125  */
126 public void setSide(char side) {
127     switch (side) {
128     case 'l':
129         controlledPlayer.setSide(Side.l);
130         break;
131     case 'r':
132         controlledPlayer.setSide(Side.r);
133         break;
134     }
135 }
136
137 /**
138  * Updates the ViewedObjects of the player controlled by the
139  * Client
140  * @param objects An array of objects seen by the Client
141  */
142 public void see(int cycle, ViewedObject[] objects) {
143     lastSeeCycle = cycle; // Used for deciding when next
144     command should be sent
145     this.cycle = cycle;
146     controlledPlayer.see(cycle, objects);
147 }
148
149 /**
150  * Registers messages received from teammates
151  * @param objects An array of objects seen by the Client
152  */
153 public void hear(int cycle, String message) {
154     controlledPlayer.hear(cycle, message);
155 }
156
157 /**
158  * Returns whether the player (thinks he) knows where the
159  * ball is
160  * @return true if the player thinks he knows where the ball
161  * is, false otherwise
162  */
163 public boolean isBallLocationKnown() {
164     return !controlledPlayer.ballGone() &&

```

```

163     cycle -
164     controlledPlayer.getWorld().getBall().getSeen() < 20;
165 }
166
167 /**
168  * Returns if the player believes that the ball is in his
169  * rectangle
170  * @return True if the player believes that the ball is in
171  * his rectangle
172  */
173 public boolean ballIsClose() {
174     return controlledPlayer.getRectangle().contains(
175         controlledPlayer.getWorld().getBall().getPositionAt(
176             cycle + 1));
177 }
178
179 /**
180  * Returns if the player is in his rectangle
181  * @return True if the player is in his rectangle
182  */
183 public boolean isInPosition() {
184     return controlledPlayer.getRectangle().contains(
185         controlledPlayer.getPosition());
186 }
187
188 /**
189  * Returns if a player is near good positioning as a
190  * goalkeeper
191  */
192 public boolean isInGKPos() {
193     return controlledPlayer.findGKPos(cycle + 1).distance(
194         controlledPlayer.getPosition()) < 2;
195 }
196
197 /**
198  * Returns if a player is at his team position (or very
199  * nearly)
200  * @return True if player is very close to his team position
201  */
202 public boolean isAtHome() {
203     return controlledPlayer.getTeamPosition().distance(
204         controlledPlayer.getPosition()) < 2;
205 }
206
207 public boolean isPassAvailable() {
208     return !controlledPlayer.getPassTarget(cycle).equals(
209         controlledPlayer.getWorld().getOppGoal().getPosition(
210             ));
211 }
212
213 public boolean isInShootingRange() {
214     return controlledPlayer.getDistTo(controlledPlayer.
215         getWorld().getOppGoal().getPosition()) < 30;
216 }
217
218
219
220

```

```

205 public boolean isNearOwnGoal() {
206     return controlledPlayer.getDistTo(controlledPlayer.
207         getWorld().getOwnGoal().getPosition()) < 20;
208 }
209
210 /**
211  * Returns if a player is very close to a player on his own
212  * team who is
213  * nearer to the ball than them
214  * @return True if the player is too close to a teammate
215  */
216 public boolean isTooClose() {
217     WorldObject[] teamMates = controlledPlayer.getWorld().
218         getOwnPlayers();
219     for (WorldObject p : teamMates) {
220         if (p != null) {
221             if (cycle - p.getSeen() < 10 &&
222                 controlledPlayer.getDistTo(p, cycle + 1) < 10 &&
223                 p.getPosition().distance(controlledPlayer.getWorld(
224                     ).getBall().getPositionAt(cycle + 1)) <
225                 controlledPlayer.getDistTo(controlledPlayer.
226                     getWorld().getBall(), cycle + 1)) {
227                 return true;
228             }
229         }
230     }
231     return false;
232 }
233
234 /**
235  * Returns if the player's team is in possession of the ball
236  * @return True if the team is in possession, false
237  * otherwise
238  */
239 public boolean isAttacking() {
240     WorldObject worldBall = controlledPlayer.getWorld().
241         getBall();
242     WorldObject[] teamMates = controlledPlayer.getWorld().
243         getOwnPlayers();
244     WorldObject[] oppPlayers = controlledPlayer.getWorld().
245         getOppPlayers();
246     float myNearest = controlledPlayer.getDistTo(worldBall,
247         cycle + 1);
248     for (WorldObject p : teamMates) {
249         if (p != null) {
250             if (cycle - p.getSeen() < 10) {
251                 myNearest = Math.min(myNearest,
252                     Coordinate.distance(p.getPositionAt(cycle + 1),
253                         worldBall.getPositionAt(cycle + 1)));
254             }
255         }
256     }
257     float theirNearest = 200;
258 }

```

```

250     for (WorldObject p : oppPlayers) {
251         if (p != null) {
252             if (cycle - p.getSeen() < 10) {
253                 theirNearest = Math.min(theirNearest,
254                     Coordinate.distance(p.getPositionAt(cycle + 1),
255                         worldBall.getPositionAt(cycle + 1)));
256             }
257         }
258     }
259     return myNearest < theirNearest;
260 }
261
262 /**
263  * Returns if the player believes that he is the closest
264  * play on his team to the ball
265  * @return True if the player believes he is closest to the
266  * ball, false otherwise
267  */
268 public boolean isMyNearest() {
269     WorldObject worldBall = controlledPlayer.getWorld().
270         getBall();
271     WorldObject[] teamMates = controlledPlayer.getWorld().
272         getOwnPlayers();
273     float myNearest = 200;
274     for (WorldObject p : teamMates) {
275         if (p != null) {
276             if (cycle - p.getSeen() < 10) {
277                 myNearest = Math.min(myNearest,
278                     Coordinate.distance(p.getPositionAt(cycle + 1),
279                         worldBall.getPositionAt(cycle + 1)));
280             }
281         }
282     }
283     return myNearest >= controlledPlayer.getDistTo(worldBall,
284         cycle + 1);
285 }
286
287 /**
288  * Returns whether the player can 'teleport'
289  * @return True if the player can 'teleport', False
290  * otherwise
291  */
292 public boolean canMove() {
293     return gameState.equals("before_kick.off") ||
294         gameState.substring(0, 6).equals("goal.l") ||
295         gameState.substring(0, 6).equals("goal.r");
296 }
297
298 /**
299  * Returns whether the ball is dead
300  */
301 public boolean isDeadBall() {

```

```

298     return !gameState.equals("play.on");
299 }
300
301 /**
302  * Returns whether it is this team's set piece
303  */
304 public boolean isOurDeadBall() {
305     String side = controlledPlayer.getSide().equals(GameObject
306         .Side.l) ? "l" : "r";
307     return isDeadBall() && gameState.substring(gameState.
308         length() - 2).equals(side);
309 }
310
311 /**
312  * Returns whether the player is close enough to the ball (
313  * and can see it) to kick it
314  * @return True if the player can kick the ball, false
315  * otherwise
316  */
317 public boolean canKickBall() {
318     WorldObject worldBall = controlledPlayer.getWorld().
319         getBall();
320     float kickableMargin = new Float(serverParams.get("
321         kickable_margin"));
322     float playerSize = new Float(serverParams.get("player_size
323         "));
324     float ballSize = new Float(serverParams.get("ball_size"));
325     return controlledPlayer.getDistTo(worldBall, cycle + 1) <=
326         kickableMargin + playerSize + ballSize &&
327         cycle - worldBall.getSeen() < 5;
328 }
329
330 /**
331  * Returns whether the player is currently in a position to
332  * catch the ball
333  * Only goalkeepers can catch the ball
334  * @return true if the player can catch the ball, false
335  * otherwise
336  */
337 public boolean canCatchBall() {
338     Coordinate ballPos = controlledPlayer.getWorld().getBall().
339         getPositionAt(cycle + 1);
340     float catchableDistance = Float.parseFloat(serverParams.
341         get("catchable_area.l"));
342     return controlledPlayer.getDistTo(ballPos) <
343         catchableDistance &&
344         controlledPlayer.isGoalkeeper() && ballPos.y < -36 &&
345         ballPos.x < 20 && ballPos.y > -20;
346 }
347
348 public boolean canSendCommand() {
349     String simulatorStep = serverParams.get("simulator_step");
350     String senseBodyStep = serverParams.get("sense_body_step");
351 }

```

```

337     if (simulatorStep == null || senseBodyStep == null) return
338         false;
339     if (simulatorStep.equals(senseBodyStep)) {
340         return ready;
341     } else {
342         if (simulatorStep != null && lastSentCommand <= System.
343             currentTimeMillis() -
344             Integer.parseInt(simulatorStep)) {
345             return true;
346         } else {
347             return false;
348         }
349     }
350     public void makeReady() {
351         ready = true;
352     }
353     /*
354     * ACTIONS
355     */
356     /**
357     * Makes the player go to their position
358     */
359     public void goHome() {
360         if (canMove()) {
361             move(controlledPlayer.getTeamPosition());
362         } else {
363             goTo(controlledPlayer.getTeamPosition());
364         }
365     }
366     public void turnToFace(Coordinate p) {
367         turn(controlledPlayer.getDirOf(p));
368     }
369     public void goTo(Coordinate p) {
370         if (lastSeeCycle - controlledPlayer.getPosCalculated() <
371             6) {
372             float angDiff = controlledPlayer.getDirOf(p);
373             if (Math.abs(angDiff) > 5) {
374                 turnToFace(p);
375             } else {
376                 dash(100);
377             }
378         } else {
379             turn(45);
380         }
381     }
382     public void faceBall() {
383         if (lastSeeCycle - controlledPlayer.getPosCalculated() <

```

29

```

388         6) {
389             turnToFace(controlledPlayer.getWorld().getBall().
390                 getPositionAt(cycle + 1));
391         } else {
392             turn(45);
393         }
394     }
395     public void runToBall(int speed) {
396         WorldObject ball = controlledPlayer.getWorld().getBall();
397         float actualSpeed = Float.parseFloat(serverParams.get("
398             player.speed.max")) / 2;
399         Coordinate intercept = controlledPlayer.findInterceptPoint
400             (ball, actualSpeed, cycle + 1);
401         goTo(intercept);
402     }
403     // Makes the player kick the ball towards opponent's goal
404     public void shoot() {
405         kick(100, controlledPlayer.getDirOf(controlledPlayer.
406             getWorld().getOppGoal().getPosition()));
407     }
408     public void callForPass() {
409         say(String.format("passtome %1$s %2$s)",
410             controlledPlayer.getPosition().x, controlledPlayer.
411             getPosition().y));
412     }
413     // Moves the player into space
414     public void findSpace() {
415         WorldObject[] teammates = controlledPlayer.getWorld().
416             getOwnPlayers();
417         Coordinate closest = new Coordinate(0, 0);
418         for (WorldObject p : teammates) {
419             if (p != null) {
420                 if (controlledPlayer.getDistTo(p, cycle + 1) <
421                     controlledPlayer.getDistTo(closest)) {
422                     closest = p.getPosition();
423                 }
424             }
425         }
426         goTo(controlledPlayer.getPosition().subtract(closest));
427     }
428     // Moves the player to a defensive position
429     public void goToDefensivePosition() {
430         WorldObject worldBall = controlledPlayer.getWorld().
431             getBall();
432         Coordinate target = new Coordinate(Math.max(
433             controlledPlayer.getRectangle().x1, controlledPlayer.
434             getRectangle().x2), controlledPlayer.getTeamPosition()
435             .y);
436         target.x = Math.max(Math.min(target.x, worldBall.

```

```

    getPosition().x - 10, -57);
430     goTo(target);
431 }
432
433 // Moves the player to 'hold up' opposition by standing in
    front of the ball
434 public void standUp() {
435     WorldObject worldBall = controlledPlayer.getWorld().
        getBall();
436     WorldObject myGoal = controlledPlayer.getWorld().
        getOwnGoal();
437     Coordinate target = (worldBall.getPositionAt(cycle + 1).
        subtract(myGoal.getPosition()));
438     float mag = target.distance(new Coordinate(0, 0));
439     target = worldBall.getPosition().subtract(new Coordinate((
        target.x / mag) * 10, (target.y / mag) * 10));
440     target.x = Math.min(target.x, myGoal.getPosition().x);
441     goTo(target);
442 }
443
444 // Moves the player to halfway between their position and
    the ball
445 public void goToAttackingPosition() {
446     Coordinate target = new Coordinate(0, controlledPlayer.
        getTeamPosition().y);
447     target.x = Math.min(Math.max(controlledPlayer.getRectangle
        ().x1,
448         controlledPlayer.getRectangle().x2),
449         controlledPlayer.getOffsideLine()) - 5;
450     goTo(target);
451 }
452
453 // Moves the player to a sensible position for a goalkeeper
454 public void goToGKPos() {
455     goTo(controlledPlayer.findGKPos(cycle + 1));
456 }
457
458 public void passBall() {
459     Coordinate target = controlledPlayer.getPassTarget(cycle);
460     kick(controlledPlayer.getDistTo(target) * 3,
        controlledPlayer.getDirOf(target));
461     Coordinate ballPos = controlledPlayer.getWorld().getBall().
        getPositionAt(cycle + 1);
462     say(String.format("b%1$0+3.0f%2$0+3.0f", ballPos.x,
        ballPos.y));
463 }
464
465 // Makes the player kick the ball towards opponent's goal
466 public void dribble() {
467     kick(10, controlledPlayer.getDirOf(controlledPlayer.
        getWorld().getOppGoal().getPosition()));
468 }
469
470 // Actions sent directly to server

```

```

471
472 public void dash(int power) {
473     ready = false;
474     rcssServer.send(String.format("(dash %1$d)", power));
475     bodyMonitor.doDash();
476 }
477
478 /**
479  * Sends a turn message to the server, and updates the
    player's direction in
480  * the agent's world model
481  * @param angle The angle through which to turn
482  */
483 public void turn(float angle) {
484     ready = false;
485     int inertia = new Integer(serverParams.get("inertia_moment
        "));
486     int minmoment = new Integer(serverParams.get("minmoment"))
        ;
487     int maxmoment = new Integer(serverParams.get("maxmoment"))
        ;
488     float decay = new Float(serverParams.get("player_decay"));
489     float moment = angle * (1 + inertia * bodyMonitor.getSpeed
        () * decay);
490     moment = Math.max(minmoment, Math.min(maxmoment, moment));
491     rcssServer.send(String.format("(turn %1$f)", moment));
492     controlledPlayer.addAngle(moment / (1 + inertia *
        bodyMonitor.getSpeed() * decay), cycle + 1);
493     bodyMonitor.doTurn();
494 }
495
496 public void kick(float power, float angle) {
497     ready = false;
498     rcssServer.send(String.format("(kick %1$f %2$f)", power,
        angle));
499     float kickableMargin = Float.parseFloat(serverParams.get("
        kickable_margin"));
500     WorldObject worldBall = controlledPlayer.getWorld().
        getBall();
501     float resultPow = power * (1 - 0.25f * Math.abs(
        controlledPlayer.getDirOf(worldBall, cycle + 1)) -
502         0.25f * (controlledPlayer.getDistTo(worldBall, cycle +
        1) / kickableMargin));
503     resultPow = Math.min(Float.parseFloat(serverParams.get("
        ball_accel_max")), resultPow);
504     float resultAngle = controlledPlayer.getDirection() -
        angle;
505     Coordinate accel = new Coordinate(resultPow * Math.cos(
        Math.toRadians(resultAngle)),
506         resultPow * Math.sin(Math.toRadians(resultAngle)));
507     Coordinate newVel = worldBall.getVelocity().add(accel);
508     float mag = new Coordinate(0, 0).distance(newVel);
509     float maxSpeed = Float.parseFloat(serverParams.get("
        ball_speed_max"));

```

```

510     if (mag > maxSpeed) {
511         newVel = new Coordinate(newVel.x / mag, newVel.y / mag).
            times(maxSpeed);
512     }
513     worldBall.setVelocity(newVel);
514     bodyMonitor.doKick();
515 }
516
517 /**
518  * Makes the player 'teleport' to the given location (if
        allowed)
519  * @param destination The position to move to
520  */
521 public void move(Coordinate destination) {
522     ready = false;
523     rcssServer.send(String.format("(move %1$f %2$f)",
524         destination.x, -destination.y));
525     controlledPlayer.setPosition(destination);
526 }
527
528 /**
529  * Makes the player catch the ball
530  */
531 public void catchBall() {
532     ready = false;
533     rcssServer.send(String.format("(catch %1$f)",
66 534         controlledPlayer.getDirOf(controlledPlayer.getWorld().
            getBall(), cycle + 1)));
535 }
536
537 /**
538  * Makes the player say something
539  * @param message What the player should say
540  */
541 public void say(String message) {
542     rcssServer.send(String.format("(say %1$s)", message.
        replace(',', ' ')));
543 }
544
545 }

```

Listing B.11: Parser.java

```

1 package behaviours;
2
3 import java.util.ArrayList;
4 import java.util.HashMap;
5
6 import world.*;
7
8 public class Parser {
9
10     private Client client; // The Client that this parser is
        for
11

```

```

12 public Parser(Client c) {
13     this.client = c;
14 }
15 /**
16  * Makes sense of the received input
17  * @param client The client that received the input
18  * @param input The input that was received
19  */
20 public void Parse(Client client, String input) {
21     String[] tokens = tokenise(input);
22     for (String t : tokens) {
23         String[] subtokens = tokenise(t);
24         if (subtokens.length == 0) return;
25         if (subtokens[0].equals("init")) {
26             client.setSide(subtokens[1].charAt(0));
27             client.setNumber(Integer.parseInt(subtokens[2]));
28             client.setGameState(subtokens[3]);
29         } else if (subtokens[0].equals("see")) {
30             client.see(Integer.parseInt(subtokens[1]), parseSee(t)
                );
31         } else if (subtokens[0].equals("hear")) {
32             parseHear(t);
33         } else if (subtokens[0].equals("server.param")) {
34             client.setServerParams(parseServerParams(t));
35         } else if (subtokens[0].equals("sense.body")) {
36             parseBody(t);
37             client.setCycle(Integer.parseInt(subtokens[1]));
38             client.makeReady();
39         }
40     }
41 }
42
43 /**
44  * Takes a 'see' string from the server, and returns an
        array of viewed
45  * objects represented in the String
46  * @param input A 'see' string as received from RCSSServer
47  * @return An array of ViewedObjects as represented by the '
        see' String
48  */
49 private ViewedObject[] parseSee(String input) {
50     String[] objectStrings = tokenise(input);
51     String[] objectStringsConcat = new String[objectStrings.
        length - 2];
52     float goalWidth = -1;
53     while (goalWidth == -1) {
54         String gw = client.getServerParams().get("goal.width");
55         if (gw != null) goalWidth = new Float(gw);
56     }
57     for (int i = 0; i < objectStringsConcat.length; i++) {
58         objectStringsConcat[i] = objectStrings[i + 2];
59     }
60     objectStrings = objectStringsConcat;
61     ArrayList<ViewedObject> objectArrayList = new ArrayList<

```

```

        ViewedObject>());
    for (String os : objectStrings) {
    62 String[] objectDetailStrings = tokenise(os);
    63 char objectChar = objectDetailStrings[0].charAt(0);
    64 GameObject gameObject;
    65 float distance = 0;
    66 float direction = 0;
    67 if (objectDetailStrings.length > 2) {
    68     distance = Float.parseFloat(objectDetailStrings[1]);
    69     direction = Float.parseFloat(objectDetailStrings[2]);
    70 } else {
    71     direction = Float.parseFloat(objectDetailStrings[1]);
    72 }
    73 switch (objectChar) { // Looks at the type of object
    74 case 'p':
    75 case 'P':
    76     gameObject = new Player(tokenise(objectDetailStrings
    77         [0]), client.getTeamName());
    78     break;
    79 case 'b':
    80 case 'B':
    81     gameObject = new Ball();
    82     break;
    83 case 'f':
    84     gameObject = new Flag(tokenise(objectDetailStrings[0])
    85         , goalWidth,
    86         client.getControlledPlayer().getSide());
    87     break;
    88 case 'g':
    89     char side = (tokenise(objectDetailStrings[0]))[1].
    90         charAt(0);
    91     gameObject = new Goal(side);
    92     break;
    93 default:
    94     gameObject = null;
    95     break;
    96 }
    97 if (gameObject != null) {
    98     float aSpeed = 0;
    99     float dSpeed = 0;
    100     if (objectDetailStrings.length >= 5) {
    101         dSpeed = new Float(objectDetailStrings[3]);
    102         aSpeed = new Float(objectDetailStrings[4]);
    103     }
    104     ViewedObject viewedObject = new ViewedObject(
    105         gameObject, distance, direction, aSpeed, dSpeed);
    106     objectArrayList.add(viewedObject);
    107 }
    108 }
    109 return objectArrayList.toArray(new ViewedObject[0]);
    110 }

    private void parseHear(String input) {
    109 String tokens[] = tokenise(input);
    110

```

```

    111 if (tokens[2].equals("referee")) {
    112     client.setGameState(tokens[3]);
    113 } else {
    114     if (tokens[2].equals("self")) return;
    115     if (tokens[3].equals("our"))
    116         client.hear(Integer.parseInt(tokens[1]), tokens[5]);
    117 }
    118 }
    119
    120 /**
    121  * Takes a string as received from the server of
    122     server_params
    123  * and returns a HashMap of params and values
    124  * @param paramString The string to extract parameters from
    125  * @return A HashMap with keys of parameter names and values
    126  * of parameter values
    127  */
    128 private HashMap<String, String> parseServerParams(String
    129     paramString) {
    130     HashMap<String, String> params = new HashMap<String,
    131         String>();
    132     String[] tokens = tokenise(paramString);
    133     for (int i = 1; i < tokens.length; i++) {
    134         String[] subtokens = tokenise(tokens[i]);
    135         params.put(subtokens[0], subtokens[1]);
    136     }
    137     return params;
    138 }
    139
    140 private void parseBody(String input) {
    141     String[] tokens = tokenise(input);
    142     for (String t : tokens) {
    143         String[] subTokens = tokenise(t);
    144         if (subTokens[0].equals("kick")) {
    145             client.getBodyMonitor().setKick(new Integer(subTokens
    146                 [1]));
    147         } else if (subTokens[0].equals("dash")) {
    148             client.getBodyMonitor().setDash(new Integer(subTokens
    149                 [1]));
    150         } else if (subTokens[0].equals("turn")) {
    151             client.getBodyMonitor().setTurn(new Integer(subTokens
    152                 [1]));
    153         } else if (subTokens[0].equals("speed")) {
    154             client.getBodyMonitor().setSpeed(new Float(subTokens
    155                 [1]));
    156             client.getControlledPlayer().calculateVelocity(
    157                 Float.parseFloat(subTokens[1]),
    158                 Float.parseFloat(subTokens[2]));
    159         }
    160     }
    161 }
    162
    163 /**
    164  * Splits string into tokens separated by spaces, and

```

```

    grouped by parentheses
158 * @param input The string to tokenise
159 * @return An array of strings , one representing each token
160 */
161 private String[] tokenise(String input) {
162     ArrayList<String> returnArrayList = new ArrayList<String>
        >();
163     int stringPos = 0;
164     while (stringPos < input.length()) {
165         switch (input.charAt(stringPos)) {
166             case '(':
167                 int startingPos = stringPos + 1;
168                 int bracketCount = 1;
169                 while (bracketCount > 0) {
170                     int openedIndex = input.indexOf('(', stringPos + 1);
171                     int closedIndex = input.indexOf(')', stringPos + 1);
172                     if (openedIndex == -1) {openedIndex = Integer.
                        MAXVALUE;} //Deals with character not
173                     if (closedIndex == -1) {closedIndex = Integer.
                        MAXVALUE;} //being found
174                     stringPos = Math.min(openedIndex, closedIndex);
175                     if (input.charAt(stringPos) == '(') {
176                         bracketCount ++;
177                     } else {
178                         bracketCount --;
179                     }
180                 }
181                 returnArrayList.add(input.substring(startingPos,
                    stringPos));
182                 stringPos ++;
183                 break;
184             case ' ':
185                 stringPos ++;
186                 break;
187             default:
188                 int spacePos = (input.indexOf(' ', stringPos));
189                 int openBracketPos = (input.indexOf('(', stringPos));
190                 if (spacePos == -1) {spacePos = Integer.MAXVALUE;}
191                 if (openBracketPos == -1) {openBracketPos = Integer.
                    MAXVALUE;}
192                 spacePos = Math.min(spacePos, openBracketPos);
193                 spacePos = Math.min(spacePos, input.length());
194                 String token = input.substring(stringPos, spacePos);
195                 returnArrayList.add(token);
196                 stringPos = spacePos;
197                 break;
198             }
199         }
200     return returnArrayList.toArray(new String[0]);
201 }
202
203 }

```

Listing B.12: RCSSServer.java

```

1 package behaviours;
2
3 import java.io.IOException;
4 import java.net.DatagramPacket;
5 import java.net.DatagramSocket;
6 import java.net.InetAddress;
7 import java.net.SocketException;
8 import java.net.SocketTimeoutException;
9 import java.net.UnknownHostException;
10
11 public class RCSSServer {
12
13     private InetAddress host;
14     private int port;
15     private DatagramSocket sock;
16     private Client client;
17     private int bufferSize = 4000;
18     private long startTime;
19
20     public RCSSServer(Client c, String h, int p) {
21         client = c;
22         try {
23             host = InetAddress.getByName(h);
24         } catch (UnknownHostException e) {
25             e.printStackTrace();
26         }
27         port = p;
28         c.react(init());
29     }
30
31     public RCSSServer(Client c) {
32         this(c, "localhost", 6000);
33     }
34
35     private String init() {
36         String receivedString = null;
37         try {
38             sock = new DatagramSocket();
39             send(String.format("(init %1$s (version %2$s) %3$s)",
40                 client.getTeamName(), client.getVersion(),
41                 (client.getControlledPlayer().isGoalkeeper()) ? "(
                    goalie)" : ""));
42             byte[] receivedBytes = new byte[bufferSize];
43             DatagramPacket receivedPacket = new DatagramPacket(
                receivedBytes, receivedBytes.length);
44             sock.setSoTimeout(10000);
45             sock.receive(receivedPacket);
46             host = receivedPacket.getAddress();
47             port = receivedPacket.getPort();
48             ReceiveThread rThread = new ReceiveThread();
49             rThread.start();
50             Parser parser = new Parser(client);
51             receivedString = new String(receivedPacket.getData());

```



```

52     parser.Parse(client, receivedString);
53 } catch (SocketTimeoutException e) {
54     System.err.println("Connection to server timed out.");
55 } catch (SocketException e) {
56     e.printStackTrace();
57 } catch (IOException e) {
58     e.printStackTrace();
59 }
60 return receivedString;
61 }
62
63 public void send(String message) {
64     message = message + Character.MIN_VALUE; //RCSSS likes
        null-character-terminated messages
65     byte[] dataToSend = message.getBytes();
66     try {
67         DatagramPacket packetToSend = new DatagramPacket(
68             dataToSend, dataToSend.length, host, port);
69         sock.send(packetToSend);
70         client.updateLastSentCommand();
71     } catch (UnknownHostException e) {
72         System.err.println("Could not find localhost");
73     } catch (SocketException e) {
74         e.printStackTrace();
75     } catch (IOException e) {
76         e.printStackTrace();
77     }
78 }
79
80 private class ReceiveThread extends Thread {
81     public void run() {
82         try {
83             byte[] receivedBytes = new byte[bufferSize];
84             DatagramPacket receivedPacket = new DatagramPacket(
85                 receivedBytes, receivedBytes.length);
86             while (true) {
87                 sock.receive(receivedPacket);
88                 String receivedString = new String(receivedPacket.
89                     getData());
90                 client.react(receivedString.trim());
91             }
92         } catch (SocketTimeoutException e) {
93             System.err.println("Connection to RCSS Server timed
94                 out.");
95         } catch (Exception e) {
96             e.printStackTrace();
97         }
98     }
99 }

```

Listing B.13: ControlledPlayer.java

```

1 package world;
2

```

```

3 import behaviours.Client;
4 import java.util.ArrayList;
5 import java.util.Collections;
6 import java.lang.Math;
7
8 public class ControlledPlayer extends Player {
9
10     private Coordinate position = new Coordinate(0,0);
11     private float direction = 0;
12     private float viewDirection = 0;
13     // The cycle in which the player's position was last
        calculated
14     private int posCalculated = 0;
15     private ViewedObject[] viewedObjects = new ViewedObject[0];
16     private Coordinate teamPosition = new Coordinate(0, 0);
17     private Coordinate velocity = new Coordinate(0, 0);
18     private Rectangle rectangle = new Rectangle(-57, 57, -39,
        39);
19     private World world = new World(this);
20     private Side side;
21     private boolean goalkeeper = false;
22     private Client client;
23
24     public ControlledPlayer(Client client) {
25         this.client = client;
26     }
27
28     public World getWorld() {
29         return world;
30     }
31
32     public ViewedObject[] getViewedObjects() {
33         return viewedObjects;
34     }
35
36     public Coordinate getPosition() {
37         return position;
38     }
39
40     public void setPosition(Coordinate position) {
41         this.position = position;
42     }
43
44     public float getDirection() {
45         return direction;
46     }
47
48     public int getPosCalculated() {
49         return posCalculated;
50     }
51
52     public Coordinate getTeamPosition() {
53         return teamPosition;
54     }
55 }

```

```

55
56 public void setTeamPosition(Coordinate teamPosition) {
57     this.teamPosition = teamPosition;
58 }
59
60 public void setGoalkeeper(boolean goalkeeper) {
61     this.goalkeeper = goalkeeper;
62 }
63
64 public boolean isGoalkeeper() {
65     return goalkeeper;
66 }
67
68 public Rectangle getRectangle() {
69     return rectangle;
70 }
71
72 public void setRectangle(Rectangle rectangle) {
73     this.rectangle = rectangle;
74 }
75
76 public Client getClient() {
77     return client;
78 }
79
80 public Side getSide() {
81     return side;
82 }
83
84 public void setSide(Side side) {
85     this.side = side;
86 }
87
88 public void see(int cycle, ViewedObject[] os) {
89     viewedObjects = os;
90     Coordinate newPos = calculatePosition(cycle, os);
91     float newDir = calculateDirection(newPos, os);
92     if (cycle >= this.posCalculated) {
93         if (getFlags(os).size() >= 3) {
94             this.posCalculated = cycle;
95         }
96         this.position = newPos;
97         this.direction = newDir;
98         this.viewDirection = newDir;
99     }
100     for (ViewedObject o : viewedObjects) {
101         // Calculates position
102         Coordinate pos = new Coordinate(
103             newPos.x + o.getDistance() *
104             Math.cos(Math.toRadians(newDir - o.getDirection())),
105             newPos.y + o.getDistance() *
106             Math.sin(Math.toRadians(newDir - o.getDirection()))
107         );
108         // Calculates velocity, if there is one

```

```

108 Coordinate absvel = new Coordinate(0, 0);
109 if (o.getSpeed() != 0 && o.getdSpeed() != 0) {
110     // Calculates relative velocity in Cartesian
111     // coordinates
112     Coordinate relvel = new Coordinate(
113         o.getdSpeed() * Math.cos(Math.toRadians(o.
114             getaSpeed())),
115         o.getdSpeed() * Math.sin(Math.toRadians(o.
116             getaSpeed())));
117     // Calculates absolute velocity by rotating then
118     // adding own velocity
119     float radDirection = (float) Math.toRadians(newDir);
120     absvel = new Coordinate(
121         relvel.x * Math.cos(2 * radDirection) +
122         relvel.y * Math.sin(2 * radDirection) + velocity.x
123         ,
124         relvel.x * Math.sin(2 * radDirection) -
125         relvel.y * Math.cos(2 * radDirection) + velocity.y
126     );
127 }
128 if (o.getObject() instanceof Ball) {
129     world.getBall().setPosition(pos);
130     world.getBall().setSeen(posCalculated);
131     world.getBall().setVelocity(absvel);
132 } else if (o.getObject() instanceof Player) {
133     world.processPlayer(new WorldObject(world, o.getObject()
134         (),
135         pos, posCalculated, absvel));
136 }
137 }
138 }
139
140 public void hear(int cycle, String message) {
141     switch (message.charAt(0)) {
142     case 'b':
143         if (world.getBall().getSeen() - cycle > 3) {
144             float xpos = Float.parseFloat(message.substring(1, 3))
145             ;
146             float ypos = Float.parseFloat(message.substring(4, 3))
147             ;
148             world.getBall().setPosition(new Coordinate(xpos, ypos)
149             );
150             world.getBall().setSeen(cycle);
151         }
152     }
153 }
154
155 // Takes the polar velocity, and sets absolute velocity
156 // based on this
157 public void calculateVelocity(float speed, float dir) {
158     velocity = new Coordinate(
159         speed * Math.cos(Math.toRadians(this.direction - dir))
160         ,
161         speed * Math.sin(Math.toRadians(this.direction - dir))

```

```

150     );
151 }
152 public ViewedObject getViewedBall() {
153     ViewedObject vBall = null;
154     ViewedObject[] viewedObjectsCache = viewedObjects;
155     int i = viewedObjectsCache.length - 1;
156     while (vBall == null && i >= 0) {
157         if (viewedObjectsCache[i].getObject() instanceof Ball) {
158             vBall = viewedObjectsCache[i];
159         } else {
160             i--;
161         }
162     }
163     return vBall;
164 }
165
166 /**
167  * Returns if the player has realised that the ball's
168  * location in his world
169  * model has turned out to be wrong, based on what he has
170  * seen
171  * @return
172  */
173 public boolean ballGone() {
174     float visibleAngle =
175         Float.parseFloat(client.getServerParams().get("
176             visible.angle"));
177     boolean shouldSee =
178         Math.abs(getViewedDirOf(getWorld().getBall(),
179             posCalculated) - viewDirection)
180         < visibleAngle / 2;
181     return shouldSee && getViewedBall() == null;
182 }
183
184 /**
185  * Alters the player's direction by the specified angle, for
186  * the specified cycle
187  */
188 public void addAngle(float angle, int cycle) {
189     posCalculated = cycle;
190     this.direction = World.boundedAngle(this.direction - angle
191 );
192 }
193
194 // Returns the direction of a given coordinate relative to
195 // the controlled player,
196 // based on the most up-to-date position calculations
197 public float getDirOf(Coordinate coords) {
198     float dir = (float) Math.toDegrees(Math.atan2((coords.y -
199         position.y),
200         (coords.x - position.x)));
201     return World.boundedAngle(direction - dir);
202 }

```

```

195
196 // Returns the direction of a given WorldObject relative to
197 // the controlled player
198 public float getDirOf(WorldObject obj, int cycle) {
199     return getDirOf(obj.getPositionAt(cycle));
200 }
201
202 // Returns the direction of a given coordinate relative to
203 // the controlled player,
204 // based on the direction he was facing at the last visual
205 // information
206 public float getViewedDirOf(Coordinate coords) {
207     float dir = (float) Math.toDegrees(Math.atan2((coords.y -
208         position.y),
209         (coords.x - position.x)));
210     return World.boundedAngle(viewDirection - dir);
211 }
212
213 // Returns the direction of a given coordinate relative to
214 // the controlled player,
215 // based on the direction he was facing at the last visual
216 // information
217 public float getViewedDirOf(WorldObject obj, int cycle) {
218     return getViewedDirOf(obj.getPositionAt(cycle));
219 }
220
221 // Returns the distance of a given coordinate from the
222 // controlled player
223 public float getDistTo(Coordinate coord) {
224     return position.distance(coord);
225 }
226
227 // Returns the distance to a given WorldObject
228 public float getDistTo(WorldObject obj, int cycle) {
229     return getDistTo(obj.getPositionAt(cycle));
230 }
231
232 /**
233  * Given an array of ViewedObjects, returns those that are
234  * flags
235  * @param viewedObjects
236  * @return the Flags
237  */
238 private ArrayList<ViewedObject> getFlags(ViewedObject[]
239     viewedObjects) {
240     ArrayList<ViewedObject> flags = new ArrayList<ViewedObject>
241         >();
242     for (ViewedObject i : viewedObjects) {
243         if (i.getObject() instanceof Flag) {
244             flags.add(i);
245         }
246     }
247     return flags;
248 }

```

```

239
240 /**
241  * Calculates the player's position, from the objects he is
    viewing
242  * @param cycle
243  * @param viewedObjects
244  */
245 private Coordinate calculatePosition(int cycle, ViewedObject
    [] viewedObjects) {
246     Coordinate result = this.position;
247     ArrayList<ViewedObject> flags = getFlags(viewedObjects);
248     if (flags.size() >= 3) {
249         Collections.sort(flags);
250         Coordinate pos0 = ((Flag)(flags.get(0).getObject())).
            position;
251         Coordinate pos1 = ((Flag)(flags.get(1).getObject())).
            position;
252         Coordinate pos2 = ((Flag)(flags.get(2).getObject())).
            position;
253         double d = pos0.distance(pos1);
254         double d1 = (Math.pow(flags.get(0).getDistance(), 2)
255             - Math.pow(flags.get(1).getDistance(), 2) + Math.pow
                (d, 2)) / (d * 2);
256         double h = Math.sqrt(Math.pow(flags.get(0).getDistance()
            , 2) -
                Math.pow(d1, 2));
257         if (Double.isNaN(h)) h = 0;
258         double x3 = pos0.x + (d1 * (pos1.x - pos0.x)) / d;
259         double y3 = pos0.y + (d1 * (pos1.y - pos0.y)) / d;
260         double x4.i = x3 + (h * (pos1.y - pos0.y)) / d;
261         double y4.i = y3 - (h * (pos1.x - pos0.x)) / d;
262         double x4.ii = x3 - (h * (pos1.y - pos0.y)) / d;
263         double y4.ii = y3 + (h * (pos1.x - pos0.x)) / d;
264         Coordinate posi = new Coordinate((float)x4.i, (float)
            y4.i);
265         Coordinate posii = new Coordinate((float)x4.ii, (float)
            y4.ii);
266         if (Math.abs(Math.abs(posi.distance(pos2) - flags.get(2)
            .getDistance()) -
                (Math.abs(posii.distance(pos2) - flags.get(2).
                    getDistance())) < 1) {
267             // The case where the three circles intersect twice
268             if (posi.distance(position) < posii.distance(position)
                ) {
269                 result = posi;
270             } else {
271                 result = posii;
272             }
273         } else if (Math.abs(posi.distance(pos2) - flags.get(2).
            getDistance()) <
                (Math.abs(posii.distance(pos2) - flags.get(2).
                    getDistance())) &&
274             posi.x >= -57 && posi.x <= 57 && posi.y >= -39 &&
275             posi.y <= 39) {

```

```

276
277
278     // If third circle intersects with posi
279     result = posi;
280 } else {
281     // If third circle intersects with posii
282     result = posii;
283 }
284 }
285 return result;
286 }
287
288 private float calculateDirection(Coordinate position,
    ViewedObject[] viewedObjects) {
289     ArrayList<ViewedObject> flags = getFlags(viewedObjects);
290     if (flags.size() > 0) {
291         Collections.sort(flags);
292         Coordinate posn = ((Flag)(flags.get(flags.size() - 1).
            getObject())).position;
293         float dirn = flags.get(flags.size() - 1).getDirection();
294         float dir = (float)Math.toDegrees(Math.atan2(posn.y -
            position.y, posn.x - position.x)) + dirn;
295         return World.boundedAngle(dir);
296     } else {
297         return this.direction;
298     }
299 }
300
301 // Calculates where the player can intercept a given world
    object
302 public Coordinate findInterceptPoint(WorldObject obj, float
    mySpeed, int cycle) {
303     int t = 1;
304     while (t < 100) {
305         // Calculates the position of the object at t
306         Coordinate objPos = obj.getPositionAt(cycle + t);
307         if ((objPos.distance(position)) < mySpeed * t || t > 50)
            {
308             return objPos;
309         } else {
310             if (t < 10) {
311                 t += 1;
312             } else if (t < 30) {
313                 t += 5;
314             } else {
315                 t += 10;
316             }
317         }
318     }
319     return null;
320 }
321
322 // Calculates a sensible position for a goalkeeper to
    position
323 // himself at a given cycle
324 public Coordinate findGKPos(int cycle) {

```

```

325     float goalWidth = Float.parseFloat(client.getServerParams
326         ().get("goal.width"));
327     WorldObject worldBall = world.getBall();
328     WorldObject myGoal = world.getOwnGoal();
329     Coordinate target = (myGoal.getPosition().subtract(
330         worldBall.getPositionAt(cycle))).times(0.5f);
331     float mag = target.distance(new Coordinate(0, 0));
332     return myGoal.getPosition().subtract(new Coordinate((
333         target.x / mag) * (goalWidth / 2), (target.y / mag) *
334         (goalWidth / 2)));
335 }
336
337 // Returns the coordinates of a player to pass to
338 public Coordinate getPassTarget(int cycle) {
339     WorldObject[] players = world.getOwnPlayers();
340     ArrayList<WorldObject> confidentPlayers = new ArrayList<
341         WorldObject>();
342     for (WorldObject p : players) {
343         if (p != null) {
344             if (posCalculated - p.getSeen() < 10 &&
345                 p.getPosition().x > -45) { // Don't pass too far
346                 back
347                 confidentPlayers.add(p);
348             }
349         }
350     }
351     Coordinate best;
352     if (confidentPlayers.size() > 0) {
353         best = confidentPlayers.get(0).getPosition();
354     } else {
355         return world.getOppGoal().getPosition();
356     }
357     for (WorldObject p : confidentPlayers) {
358         if (p.getPosition().x > best.x) {
359             best = p.getPositionAt(cycle);
360         }
361     }
362     return best;
363 }
364
365 // Calculates the x coordinate of the opponent's offside
366 line
367 public double getOffsideLine() {
368     try {
369         double result = 0; // The lowest the offside line can be
370         is halfway
371         WorldObject[] opponents = world.getOppPlayers();
372         WorldObject deepest1 = null;
373         WorldObject deepest2 = null;
374         for (WorldObject p : opponents) {
375             if (p != null && p.getPosition().x > result) {
376                 deepest1 = p;
377                 result = deepest1.getPosition().x;
378             }
379         }
380     }

```

```

371     }
372     if (deepest1 == null) return 0;
373     result = 0;
374     for (WorldObject p : opponents) {
375         if (p != null && p.getPosition().x > result && !p.
376             getPosition().equals(deepest1.getPosition())) {
377             deepest2 = p;
378             result = deepest2.getPosition().x;
379         }
380     }
381     return Math.max(result, world.getBall().getPosition().x);
382 } catch (NullPointerException e) {
383     e.printStackTrace();
384 }
385 return 2;
386 }
387 }

```