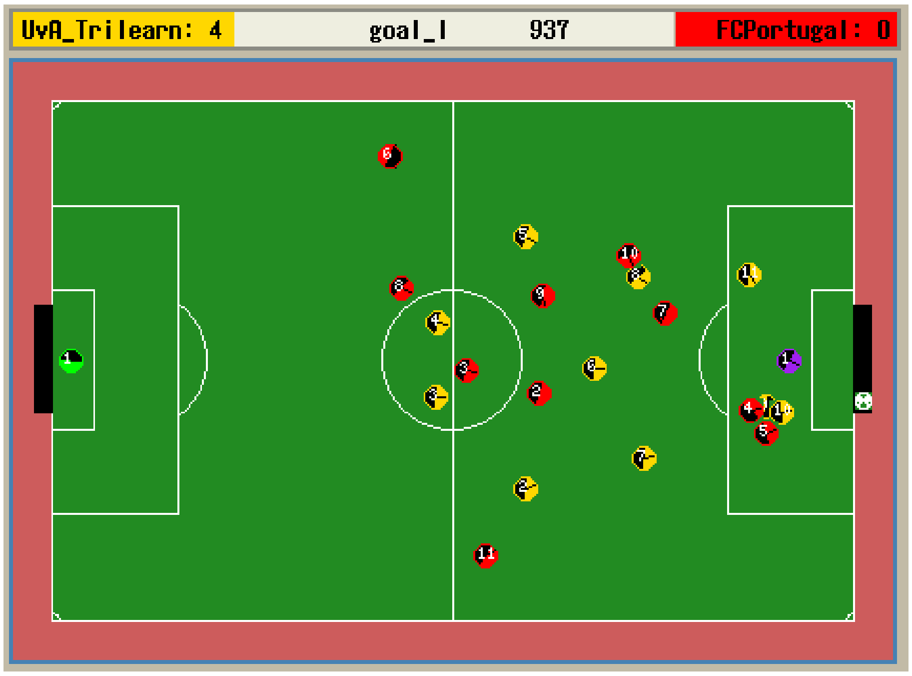
The Incremental Development of a Synthetic Multi-Agent System: The UvA Trilearn 2001 Robotic Soccer Simulation Team

Remco de Boer & Jelle Kok

February 28, 2002



Master's thesis for Artificial Intelligence and Computer Science

Faculty of Science
University of Amsterdam

Abstract

This thesis describes the incremental development and main features of a synthetic multi-agent system called *UvA Trilearn 2001*. *UvA Trilearn 2001* is a robotic soccer simulation team that consists of eleven autonomous software agents. It operates in a physical soccer simulation system called *soccer server* which enables teams of autonomous software agents to play a game of soccer against each other. The *soccer server* provides a fully distributed and real-time multi-agent environment in which teammates have to cooperate to achieve their common goal of winning the game. The simulation models many real-world complexities such as noise in object movement, noisy sensors and actuators, limited physical abilities and restricted communication. This thesis addresses the various components that make up the *UvA Trilearn 2001* robotic soccer simulation team and provides an insight into the way in which these components have been (incrementally) developed. Our main contributions include a multi-threaded three-layer agent architecture, a flexible agent-environment synchronization scheme, accurate methods for object localization and velocity estimation using particle filters, a layered skills hierarchy, a scoring policy for simulated soccer agents and an effective team strategy. Ultimately, the thesis can be regarded as a handbook for the development of a complete robotic soccer simulation team which also contains an introduction to robotic soccer in general as well as a survey of prior research in soccer simulation. As such it provides a solid framework which can serve as a basis for future research in the field of simulated robotic soccer. Throughout the project *UvA Trilearn 2001* has participated in two international robotic soccer competitions: the team reached 5th place at the *German Open 2001* and 4th place at the official *RoboCup-2001* world championship.

Acknowledgements

First of all the authors would like to thank each other for making their master's graduation project a successful and enjoyable experience. Our cooperation has always been very pleasant (even under stressful circumstances or after an inevitable setback) and has enabled us to achieve the best of our abilities. The research that is described in this thesis has truly been a team effort and without either one of us the result would certainly not have been the same. Furthermore, we thank our direct supervisor Nikos Vlassis for his comments and suggestions regarding our research and for helping us with some of the difficult problems. Here we need to mention that Nikos also deserves credit for proofreading our thesis and for helping us on the right track towards finding a solution to the optimal scoring problem. Especially our dicussion at the top of Seattle's Space Needle proved to be very valuable in this respect. Special thanks also go out to our professor Frans Groen for assigning us to the project and for guiding us through its initial stages and to Elsevier Science for sponsoring our trip to Seattle. We would also like to thank the following people for their friendship and support during our years as master's students at the University of Amsterdam: Eugene Tuinstra, Eelco Schatborn, Tijs v.d. Storm, Mart de Graaf, Mans Scholten, Martin Goede, Casper Kaandorp, Matthijs Spaan, Barry Koopen, Manuel de Vries and Claudius Blokker. Your continuing presence during lectures and practical sessions and the many enjoyable hours that you have given us away from the faculty have made our effort of graduating worthwhile. Outside the university we also owe thanks to our many close friends and relatives who have supported us throughout. Although we cannot mention all of them, the first author (RdB) would especially like to thank his mother Elly Tjoa for checking the spelling of parts of this thesis and most of all for her incredible support throughout the entire effort. Equally special thanks go out to his girlfriend Tessa Dirks who has gone through the entire graduation experience with him and who has been very understanding and supportive at all times. Without you two this thesis would surely never have happened! The second author (JK) would also like to thank his parents Jan and Henny, girlfriend Barbara, little brother Sam, sister Laura and close friends Rose, Meryam and Matthijs for their equal support and encouragement. Without a doubt all of you deserve credit for our graduation.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

In the eyes of many people soccer is not a game: it is a way of life! Although we do not quite share this view, it cannot be denied that the game of soccer plays a prominent role in the society of today. Despite this, it is probably not very common to find two students that write a master's thesis which for a large part revolves around the subject. In the same way it is probably not very common to think about soccer as a game that can be played by robots. Nevertheless, robotic soccer is a subject which in the scientific community has gained in popularity over the last five years and as such it has been the subject of our research. This first chapter provides an introduction to the subject of robotic soccer from a multi-agent perspective. The chapter is organized as follows. Section 1.1 describes the Robot World Cup (RoboCup) Initiative and its ultimate long-term goal. In Section 1.2 we then discuss the subject of robotic soccer from a multi-agent perspective. The main objectives of our research as well as the general approach that we have followed to achieve them are presented in Section 1.3. The chapter is concluded in Section 1.4 with an overview of the contents of this thesis.

1.1 The Robot World Cup Initiative

The Robot World Cup (RoboCup) Initiative is an attempt to foster artificial intelligence (AI) and intelligent robotics research by providing a standard problem where a wide range of technologies can be integrated and examined [45]. RoboCup's ultimate long-term goal is stated as follows:

> "By mid-21st century, a team of fully autonomous humanoid robot soccer players shall win a soccer game, complying with the official rules of the FIFA, against the winner of the most recent world cup for human players." [44]

It is proposed that this goal will be one of the grand challenges shared by the robotics and AI community for the next 50 years. The challenge is indeed a formidable one and given the current state of affairs in the fields of robotics and AI it sounds overly ambitious. Therefore, many people are sceptical and think that the goal will not be met. History has proven however, that human predictive powers have never been good beyond a decade. A few examples are in place here. On the 17th of December 1903, Orville Wright made the first man-carrying powered flight in an aircraft built by himself and his brother Wilbur Wright. The flight covered about 120 feet and lasted for 12 seconds [40]. If at that point someone would have claimed that roughly 66 years later the first man would set foot on the moon, he would surely have been diagnosed as mentally insane. However, on the 20th of July 1969, Neil Armstrong stepped out of the Apollo-11 Lunar Module and onto the surface of the moon [15]. Also, it took only 51 years from the release of the first operational general-purpose electronic computer in 1946[1] to the computer chess program Deep Blue which beat the human world champion in chess in 1997[2]. These examples show that many things can happen in relatively short periods of time and that one thus has to be careful when dismissing the RoboCup long-term objective as being unrealistic. There is every reason to believe however, that building a team of humanoid soccer robots will require an equally long period of time as for the previous examples.

Since it is not likely that the ultimate RoboCup goal will be met in the near future, it is important to also look for short-term objectives. In the first place, it is the intention of the RoboCup organization to use RoboCup as a vehicle to promote robotics and AI research by providing a challenging problem. RoboCup offers an integrated research task which covers many areas of AI and robotics. These include design principles of autonomous agents, multi-agent collaboration, strategy acquisition, real-time reasoning, reactive behavior, real-time sensor fusion, learning, vision, motor control, intelligent robot control, and many more [46]. In order for a humanoid robot team to actually perform a soccer game, a number of technological breakthroughs must be made and all these technologies must be incorporated. The development of these technologies can be seen as the short-term objective of the RoboCup project and even when the main goal is never achieved, several technological advancements will emerge from the effort to get there. A second intention of the RoboCup organization is to use RoboCup for educational purposes and to stimulate the interest of the general public for robotics and AI by setting forth an exciting and broadly appealing long-term objective. Currently, it seems that this intention has already succeeded. An increasing number of universities all over the world organize study projects which are related to the different aspects of RoboCup. Furthermore, the interest from the media and the general public has been increasing in successive RoboCup competitions held in recent years.

Another aspect of RoboCup is that it provides a standard problem for the evaluation of various theories, algorithms and architectures. Using a standard problem for this purpose has the advantage that different approaches can be easily compared and that progress can be measured. Computer chess is a typical example of such a standard problem which has been very successful. It has mainly been used for the evaluation and development of different search algorithms. One of the most important reasons for the success of computer chess as a standard problem has been that the strength of a computer chess program could be clearly defined by its Elo rating[3]. As a result, progress in the domain could be easily measured via actual games against human players. This is not (yet) the case for robotic soccer. With the accomplishment of Deep Blue in 1997 however, computer chess has achieved its long-term objective. The AI community therefore needs a new challenge problem and there is now general agreement that robotic soccer is suitable as a next long-range target. The main reason for this agreement is that the domain characteristics of robotic soccer are in sharp contrast to those of computer chess, as is illustrated in Table 1.1, and are considered to generate technologies which are important for the next generation of industries.

In order to achieve the RoboCup long-term objective, the RoboCup organization has introduced several robotic soccer leagues which each focus on different abstraction levels of the overall problem. Currently, the most important leagues are the following:

- **Middle Size Robot League (F-2000)**. In this league each team consists of a maximum of four robots, which are about 75cm in height and 50cm in diameter. The playing field is approximately 9 by 5 meters and the robots have no global information about the world. Important research areas for this league include localization, vision, sensor fusion, robot motor control and hardware issues.

[1]The ENIAC was built by J. Presper Eckert and John Mauchly at the University of Pennsylvania [67].
[2]In May 1997, Deep Blue beat Gary Kasparov 3.5-2.5 over 6 matches [86].
[3]The most common rating system used for chess players is called the Elo system, which is named after its inventor [29].

	Computer chess	Robotic soccer
Environment	static	dynamic
State change	turn-taking	real-time
Information accessibility	complete	incomplete
Sensor readings	symbolic	non-symbolic
Control	central	distributed

Table 1.1: Domain characteristics of computer chess compared to those of robotic soccer. From [44].

- **Small Size Robot League (F-180)**. In this league each team consists of five robots, which are about 20cm in height and 15cm in diameter. The playing field has the size of a table-tennis table and an overhead camera provides a global view of the world for each robot. Research areas which are important for this league include intelligent robot control, image processing and strategy acquisition.
- **Sony Legged Robot League**. In this league each team consists of three Sony quadruped robots (better known as AIBOs). The playing field is similar in size to that for the Small Size League. The robots have no global view of the world but use various colored landmarks which are placed around the field to localize themselves. The main research areas for this league are intelligent robot control[4] and the interpretation of sensory information[5].
- **Simulation League**. In this league each team consists of 11 synthetic (software) agents which operate in a simulated environment. Research areas which are being explored in this league include machine learning, multi-agent collaboration and opponent modeling. Currently, the simulation league is by far the largest due to the fact that no expensive hardware is needed to build the team. Furthermore, it is much easier (and cheaper) to test a simulation team against different opponents.

It is the intention of the RoboCup organization to introduce a **RoboCup Humanoid League** for the first time at the *RoboCup-2002* robotic soccer world championship in Fukuoka (Japan). For a detailed account of the different RoboCup leagues and plans for future leagues, we refer to [108].

We will mainly concentrate on the *RoboCup Simulation League* throughout this thesis. The simulation league is based on a soccer simulation system called the *RoboCup Soccer Server* [32]. This system enables teams of autonomous software agents to play a game of soccer against each other. The *soccer server* provides a multi-agent environment in which everything happens in real time and where sensing and acting are asynchronous. Various forms of uncertainty are added into the simulation such as sensor and actuator noise, noise in object movement, limited perception, unreliable low-bandwidth communication and limited physical ability. One of the advantages of the *soccer server* is the abstraction made, which relieves researchers from having to handle robot problems such as object recognition and movement. This abstraction makes it possible to focus on higher level concepts such as learning and strategic reasoning.

1.2 Robotic Soccer from a Multi-Agent Perspective

Distributed Artificial Intelligence is a subfield of AI which is concerned with systems that consist of *multiple* independent entities that interact in a domain. Traditionally, this field has been broken into two

[4] AIBOs have as many as 20 degrees of freedom.

[5] AIBOs have 7 different types of sensors: an image sensor, an audio sensor, a temperature sensor, an infrared distance sensor, an acceleration sensor, pressure sensors (head, back, chin and legs) and a vibration sensor.

subdisciplines: Distributed Problem Solving (DPS) and Multi-Agent Systems (MAS) [6]. DPS focuses on information management issues, such as task decomposition and solution synthesis, in systems consisting of several components which work together towards a common goal. MAS on the other hand aims to provide principles for the construction of complex systems containing multiple independent agents and focuses on behavior management issues (e.g. coordination of behaviors) in such systems [101]. Since robotic soccer is an example of a multi-agent domain, we are mainly interested in the latter of these two subdisciplines throughout this thesis.

An *agent* can be seen as anything that is situated in an environment and that perceives this environment through sensors and acts upon it through effectors [84]. Besides this, the agent might have some additional knowledge about the domain or possess several sophisticated cognitive capabilities. Often, the agent also has a goal which he tries to achieve. When multiple agents reside in the same environment this is called a *multi-agent system*. The difference between multi-agent systems and single-agent systems is that multi-agent systems consist of several agents which model each other's goals and actions. From an individual agent's perspective, the main difference is that other agents can affect the dynamics of a multi-agent environment in an unpredictable way. Furthermore, the agents in a multi-agent system might interact directly in the form of communication. When a group of agents in a multi-agent system have the same long-term goal, they can be regarded as a team. In order to achieve this goal, the agents must coordinate their behaviors (e.g. through communication). They must be able to act effectively both autonomously and as part of the team. In case that the environment also contains other agents which have goals that are incompatible with the common team goal, these other agents are the team's adversaries.

For general applications the use of MAS in the design of complex systems offers several advantages. Some domains even require the use of MAS as a discipline. For example, in cases where there are different entities (think of people, organizations, etc.) with different (possibly conflicting) goals and proprietary information, a multi-agent system is necessary to model their interactions [101]. But even in domains which do not necessarily require MAS, their use can bring several advantages:

- The presence of multiple agents can provide a method for *parallel* computation, thereby speeding up the operation of the system. This is especially the case for domains in which the overall task can be broken into several independent subtasks that can be handled by separate agents.
- A multi-agent system usually has a high degree of *robustness*. In systems controlled by a single entity, a single failure can cause the entire system to crash. Multi-agent systems on the other hand are said to degrade gracefully: if one or several agents fail, the system will still be operational.
- Multi-agent systems are inherently *modular* leading to simpler programming. Programmers can identify subtasks and assign control of those subtasks to different agents. This is usually easier than using a centralized agent for the whole task although for some applications this is more natural (e.g. when actions cannot be executed in parallel because the output of one is input for the other).
- The modularity of multi-agent systems enables one to add new agents to the system when necessary. This is called *scalability*. Adding new capabilities to a monolithic system is not so easy however.
- An advantage of multi-agent systems over single-agent systems is that a multi-agent system can observe the environment and perform actions in the environment at multiple locations simultaneously. It is said that a multi-agent system can take advantage of *geographical distribution* [101].
- Multi-agent systems usually have a higher *performance-cost ratio* than single-agent systems. A single robot with all the necessary capabilities for accomplishing a task is often much more expensive than the use of multiple (cheaper) robots which each have a subset of these capabilities.

From the viewpoint of Distributed Artificial Intelligence a robotic soccer game is a specific but very attractive multi-agent environment in which many interesting research issues arise [45]. In a robotic soccer game there are two competing teams. Each team consists of multiple agents that have to work together to achieve a common goal: winning the game. To fulfill this goal the team needs to score and this can be seen as a subgoal. In order to achieve this subgoal, each agent must behave quickly, flexibly and cooperatively by taking local and global situations into account. This means that although perception and action are local for each agent, they should also be part of a larger collaborative plan which is shared by all the teammates. Since the goals of both competing teams are incompatible, the opponent team can be seen as a dynamic and obstructive environment which might disturb the achievement of the common team goal. This makes the domain *collaborative* and *adversarial* at the same time [90]. Another interesting characteristic of robotic soccer is that the domain is highly *dynamic* and requires *real-time* decision making since success depends on acting quickly in response to the dynamically changing environment. Furthermore, the agents cannot accurately perceive or affect the world due to sensor and actuator noise. In addition, they have to deal with the fact that large parts of the state space are unobserved ('hidden') because their perception range is limited.

All the characteristics of robotic soccer described above also apply in simulated robotic soccer. This too is a fully distributed multi-agent domain with both teammates and adversaries. The *RoboCup Soccer Server* models many real-world complexities such as noise in object movement, noisy sensors and actuators, limited physical ability and restricted communication. Agents must respond to events and make their decisions in real time. They only have a partial view of the world at any moment which causes large parts of the state space to remain hidden from them. In addition, the perception and action cycles in the simulation are *asynchronous* which makes it impossible to rely on the traditional AI paradigm of using perceptual input to trigger actions. An agent also has only limited information about environmental state transitions resulting from the fact that the actions performed by teammates and opponents are unknown to him [103]. Since the state space of a soccer game is enormous and too large to hand-code all possible situations and agent behaviors, it is essential that agents *learn* to play the game strategically. Simulation soccer from a multi-agent perspective is a very suitable domain for research in this direction.

1.3 Main Objectives and Approach

In the past, the University of Amsterdam has been successful in the *RoboCup Simulation League* with the team *Windmill Wanderers* [17, 18], which became third at the world championship in 1998. Sadly the creator of this team, Emiel Corten, died in 1999 and as a result the soccer simulation project came to a halt. Its revival came in the autumn of the year 2000 when we started our master's graduation project on simulated robotic soccer. The main objective of the project was twofold. Firstly, we had to restart the soccer simulation project and provide a solid foundation for it which would enable others to continue the effort after our graduation. Secondly, we had to put up a good performance at the *RoboCup-2001* world championship held in the summer of 2001. Clearly, these two objectives were not completely compatible. Performing well at *RoboCup-2001* would mean that we had to set up a complete working team in a relatively short period of time and it would then not be feasible to complete each component of the system in an optimal way. The challenge was thus to find a satisfactory trade-off between the two.

Creating a complete multi-agent system, such as a simulated robotic soccer team, is not a straightforward task. The main difficulty arises from the fact that such a system consists of many different components which have to operate together in an appropriate way. Furthermore, building each separate component is a difficult task in itself. It is obvious that a project of this scale cannot become a success if it is not well organized. Software engineering aspects therefore play a prominent role in such an effort. It is important to set up a software architecture that allows for the various components to be combined in a modular fashion. This will make it easier to extend and debug the system and thus facilitates future use by others. Throughout the project much attention has therefore been focused on software engineering issues.

Another problem that had to be dealt with was that the competitions in which our team participated provided us with strict deadlines for producing a complete working system. This meant that we needed to have all the necessary components of the system working and successfully integrated by the time that the competitions started. However, the resources (time, manpower[6], etc.) which were available to achieve this were limited and this conflicted somewhat with our objective to set up a solid platform for the project that others could build upon. It was our initial objective to strive for optimal solutions to various subproblems of the overall problem and to provide a scientific validation for each of them. As a result, the implementation of these system components would need no future alterations and this would enable our successors to concentrate mainly on other issues. However, this approach would leave us with several high-quality components and not with a complete working system that could participate at *RoboCup-2001*. It was therefore decided to implement some of the system components suboptimally and to only optimize the ones which we thought would be most crucial for the success of the team.

Our initial intention was to use the available components of the *Windmill Wanderers* team [16] as a basis and to further develop the agent control layers to create a better team. However, after studying literature on the subject of simulated robotic soccer it became clear that the *Windmill Wanderers* agent architecture was not suitable to build upon. This was mainly because it was single-threaded (which restricted the performance) and based on an old version of the *soccer server* to which new features had been added in the meantime. We therefore started to redesign the current architecture into a multi-threaded one with a more modular structure. During this 're-engineering' process we encountered several problems which were primarily caused by the fact that the existing code did not conform to regular software standards, i.e. it was not well structured and scarcely documented. Because of these difficulties we also investigated the possibility of using the low-level implementation of other previously successful teams as a basis for our team, but we soon discovered that this would give us the same problems as with the *Windmill Wanderers* code. We felt that it would be more time-consuming to reuse existing source codes and restructure them into the architecture that we desired than to write all the code ourselves. Furthermore, it was our opinion that the low-level methods used by the top teams from recent years could be improved in several ways. We therefore decided to build a new team from scratch. This would enable us to structure the code exactly as we wanted and had the additional advantage that at each moment in time we would know the complete functionality of the system[7] which would make it much easier to extend and debug the code.

The main problem when building a large system, such as a simulated robotic soccer team, is that it is too big to be completely and accurately specified in advance and too complex to be built without faults. We have therefore implemented our team according to a software development technique called *incremental development* [8, 62, 68]. This approach dictates that a system should first be made to run, even though it does nothing useful except for creating the proper set of dummy objects. Each object is then gradually refined by adding more and more functionality until the system is fully 'grown'. The main advantage of this technique was that it gave us a working system at all times (which was good for moral) that could be tested and compared to previous versions. In this way we would at least be sure to have a working team ready for the forthcoming RoboCup competitions. Furthermore, the approach made it easier to locate the faults in the system since we always knew that they had to originate from the last refinement step. We applied *incremental development* by first creating an extremely simple system that had the desired multi-threaded architecture and that could perform the basic loop of receiving information from the server, processing this information and sending an action to the server. Each component in this system was built

[6] Having more people yields several benefits, but also brings many disadvantages. This will be addressed in Appendix A.3.

[7] A problem with using large programs written by other people is that you never exactly know which functions have been implemented and how it is done.

in a simple way only performing its task at a very elementary level. Some components would even do nothing just being implemented as void subroutines taking their correct place in the overall architecture. Although this initial system clearly did not do much, it certainly did it correctly and it could be regarded as our first 'working' version. We then progressively refined this simple implementation by extending the functionality of the different components one by one while keeping the architecture as a whole intact. This has eventually led to the version of our team that participated at the *RoboCup-2001* world championship.

1.4 Guide to the Thesis

In this thesis we describe the incremental development and main features of the *UvA Trilearn 2001*[8] robotic soccer simulation team [19, 21] that we have developed for our master's graduation project. Besides a high-level description of the various aspects of this team we also present the most important details of our implementation since this is something that we found lacking in current literature. Most publications related to RoboCup only describe the main contributions of a team on an abstract level and fail to provide information concerning the implementation of these contributions. Despite the fact that some teams actually release their source code after each RoboCup tournament, this makes it difficult to find a mapping between the described methodologies and the implementation of a team. In our thesis we try to bridge this gap by providing a detailed description of each component in the *UvA Trilearn* agent architecture along with the underlying reasoning that has motivated their design. Ultimately, the thesis can be regarded as a handbook for the development of a complete robotic soccer simulation team. In combination with the source code [48] that we have released it provides a solid framework for new teams to build upon and it can serve as a basis for future research in the field of simulated robotic soccer. In the remainder of this section we present a general description of the contents of each chapter that follows.

- **Chapter 2** presents a survey of related work that has resulted from a study of literature on multi-agent systems and simulated robotic soccer teams in particular. A summary is given showing the main features of each team that was studied. In this way the reader will get an idea of the research directions that have previously been explored.

- **Chapter 3** introduces the *RoboCup Soccer Server* simulation environment which has been the setting of our research. It describes the *soccer server* in detail and as such provides the context for the rest of the thesis. Topics that will be discussed include the sensor and action models in the simulation, the object movement model and the use of heterogeneous players and the coach.

- **Chapter 4** describes the *UvA Trilearn 2001* agent architecture. The different layers that make up this architecture will be shown together with the various components of the system and the way in which these components interact.

- **Chapter 5** addresses the agent-environment synchronization problem and introduces a flexible synchronization method which provides an optimal synchronization between our agents and the simulation environment. A comparative analysis of different synchronization schemes will be presented which shows that this method clearly outperforms the alternatives.

- **Chapter 6** presents the *UvA Trilearn* agent world model which can be regarded as a probabilistic representation of the world state based on past perceptions. It contains information about all the

[8]This choice of name can be motivated as follows. The first part refers to the University of Amsterdam. The second part consists of two words: 'tri' and 'learn'. 'Tri' is derived from 'three' which is a number that has several different meanings for our team: we have three team members (two students and one supervisor), we have a three-layer agent architecture and we use three threads. 'Learn' refers to the learning aspect of the team. Although in the end we have not had enough time to use learning as much as we wanted, it was our intention to use machine learning techniques to optimize several agent behaviors.

objects on the soccer field (their positions, velocities, etc.) and various methods which use this information to derive higher-level conclusions. The different attributes which are contained in the model are described and it is shown how the model is updated upon the receival of various kinds of sensory perceptions. Especially the update methods which have been used for object localization and velocity estimation of dynamic objects will be described in some detail.

- **Chapter 7** presents the *UvA Trilearn* skills hierarchy and gives a detailed description of the various player skills which are available to the agents. Some of the player skills that will be discussed include turning towards an object, kicking the ball to a desired position on the field, intercepting the ball, dribbling with the ball, passing to a teammate, marking an opponent and goaltending.

- **Chapter 8** introduces a scoring policy for simulated soccer agents. This policy enables an agent to determine the optimal target point in the goal together with an associated probability of scoring when the ball is shot to this point in a given situation. It will be shown that this problem has a dual solution after which the underlying statistical framework for computing the scoring probability will be described. This framework is partly based on an approximate method that we have developed for learning the relevant statistics of the ball motion which can be regarded as a geometrically constrained continuous-time Markov process.

- **Chapter 9** describes the *UvA Trilearn 2001* team strategy. Topics that will be discussed include team formations, the use of heterogeneous players, a model for inter-agent communication and the action selection mechanism which the agents use to choose an appropriate action in a given situation.

- **Chapter 10** presents the results of the *UvA Trilearn 2001* soccer simulation team at two international robotic soccer competitions in which it participated. We will also discuss several advantages and disadvantages of robotic soccer competitions from a scientific perspective.

- **Chapter 11** is the final chapter of this thesis. It summarizes our main contributions and presents the most important conclusions that can be drawn from the project. In this chapter we will also outline several promising directions for future work.

- **Appendix A** addresses several software engineering aspects which have played an important role throughout the project. It specifically focuses on issues concerning the implementation of our team and shows how we have tried to avoid the problems that typically arise in large software projects. Some of the topics that will be discussed include code documentation, version management, incremental software development, manpower distribution and debugging.

Chapter 2

A Survey of Related Work

During the initial stages of the project much time was spent on studying literature on the subject of multi-agent systems (MAS) and on simulated robotic soccer in particular. This has enabled us to become familiar with the robotic soccer domain and has provided us with a great deal of knowledge that has been very useful throughout the project. In retrospect, this has been an important part of our effort and we therefore feel that it is appropriate to discuss our findings in a separate chapter. In this way the reader will get an idea of the research directions that have previously been explored. This chapter is organized as follows. In Section 2.1 we present a short survey of the main features of several soccer simulation teams that we have studied and provide references for further reading about the methods that these teams have used. For each team, these references are summarized in Section 2.2 along with an overview of the most significant results of this team in international robotic soccer competitions.

2.1 Prior Research within the Simulated Robotic Soccer Domain

Robotic Soccer was first introduced as an interesting and promising domain for AI research at the Vision Interface conference in June of 1992 [59]. The first working robotic soccer systems were also described at that time [4, 85]. Since then, robotic soccer has proved to be a particularly good domain for studying a wide variety of MAS issues and for evaluating different MAS techniques in a *direct* manner[1]. As a result, the domain has been gaining in popularity in recent years, with several international competitions taking place for real robots as well as for simulated soccer agents. Since the first competitions held in 1996 (*Pre-RoboCup-96* and *MiroSot-96*), there has been an abundance of robotic soccer related research and this has led to an immense body of literature on the subject. Although some research issues can only be studied with the real robots, there are also many issues that can be investigated in simulation soccer. Space obviously does not permit an exhaustive coverage of all the work in this area and we will therefore present a survey of prior research that is most related to this thesis. This means that we will focus on simulated robotic soccer teams which have been successful in past RoboCup competitions. An overview of the main features of these teams and the research directions that they have explored will be presented in this section. References for further reading will also be provided throughout and are summarized for each team in Table 2.3 at the end of the chapter.

[1] Different teams that use different techniques can play games against each other.

2.1.1 CMUnited

This team was created by Peter Stone at Carnegie Mellon University and has been extensively described in his PhD thesis [90]. One of the main contributions of [90] is a multi-agent machine learning paradigm called *Layered Learning*. This paradigm has been designed to enable agents to learn to work together towards a common goal in an environment that is too complex to learn a direct mapping from sensors to actuators. *Layered Learning* provides a bottom-up hierarchical approach to learning agent behaviors at various levels of the hierarchy. In this framework, the learning at each level directly affects the learning at the next higher level. A possible set of learned behavior levels that is presented in [90] is shown in Table 2.1. The bottom layer contains low-level *individual* agent skills such as ball interception. The second layer contains *multi-agent* behaviors at the level of one player interacting with another. An example is pass evaluation: when an agent is in possession of the ball and has the option of passing to a particular teammate, he must have an idea of whether this teammate will be able to successfully intercept the ball. When learning this behavior, the agents can use the learned ball-interception skill as part of the multi-agent behavior. This technique of incorporating one learned behavior as part of another is an important component of *Layered Learning*. The third layer contains *collaborative* team behaviors such as pass selection: choosing to which teammate the ball should be passed. Here the agents can use their learned pass-evaluation skill to create the input space for learning the pass-selection behavior. Subsequently, the pass-selection behavior can be used as part of the training for learning a strategic positioning behavior in the layer above. Finally, the combined strategic-positioning and pass-selection behaviors can form the input representation for learning *adversarial* behaviors, such as strategic adaptation against different types of opponents.

Layer	Strategic level	Behavior type	Example
1	robot-ball	individual	ball interception
2	one-to-one player	multi-agent	pass evaluation
3	one-to-many player	team	pass selection
4	team formation	team	strategic positioning
5	team-to-opponent	adversarial	strategic adaptation

Table 2.1: Examples of different behavior levels in robotic soccer. From [90].

Layer	Learned behavior	Learning method
1	ball interception	neural network
2	pass evaluation	decision tree
3	pass selection	TPOT-RL

Table 2.2: Learning methods used for layered learning implementation of *CMUnited*. From [90].

Early implementations of *CMUnited* actually contain only three learned subtasks corresponding to the first three layers in Table 2.1. This is shown in Table 2.2. In the bottom layer the ball-interception behavior has been learned using a neural network. The pass-evaluation behavior in the second layer has been learned using the *C4.5* decision tree algorithm (see [75]) and uses the learned ball-interception skill from the layer below [93, 97]. Subsequently, the pass-selection behavior in the third layer has been learned using a new multi-agent reinforcement learning method called *TPOT-RL*[2] with the pass-evaluation skill

[2]**T**eam-**P**artitioned **O**paque-**T**ransition **R**einforcement **L**earning: this method can be used for maximizing long-term discounted reward in multi-agent environments where the agents have only limited information about environmental state transitions [103]. Although this is considered to be one of the main contributions of [90], it has never been used in any version of *CMUnited* that actually took part in a competition. This is due to the fact that it requires more training against an opponent than is possible in such situations.

from the layer below as input. Although the subtasks in the layers above have not been implemented, it is suggested that the strategic-positioning behavior can be learned using observational reinforcement learning (see [1]) and that memory-based algorithms are suitable for learning to be strategically adaptive.

Additional important features of the *CMUnited* implementation include the following:

- The agents use a *predictive* memory that gives them a precise and accurate model of the situation on the soccer field at each moment in time and that enables them to model the unseen parts of the world in a probabilistic way [7].
- An advanced communication protocol has been implemented which enables efficient and reliable inter-agent communication despite the limited communication facilities provided by the *soccer server* [94]. This communication protocol is used to ensure team coordination.
- *CMUnited* uses a *flexible* teamwork structure in which agents have flexible roles and positions inside dynamically changing formations [98].
- The agents use a sophisticated method for determining a strategic position on the field called *SPAR*[3] [115]. When positioning themselves using SPAR, the agents use a multiple-objective function with attraction and repulsion points. In this way they maximize the distance to other players and minimize the distance to the ball and the opponent goal. SPAR is an extension of similar approaches which use potential fields for positioning in highly dynamic multi-agent domains (see [55]).
- The agents make use of several pre-defined special-purpose plays (set-plays) which can be executed in situations that occur repeatedly during a soccer game [99]. Examples of such situations are kick-offs, goal-kicks, corner-kicks, etc.
- The agents of *CMUnited* use opponent behavior models to make their decisions more adaptive for different kinds of opponents [92]. This feature has been added in 1999.

CMUnited has been the most successful soccer simulation team since the official RoboCup competitions started. The team reached 4th place at *RoboCup-97* and became world champion at *RoboCup-98*. At *RoboCup-99*, *CMUnited-98*[4] reached 9th place and *CMUnited-99* became 1st again. In the year 2000, *CMUnited-99* still managed 4th place, whereas the new team *ATT-CMU-2000* finished 3rd.

2.1.2 Essex Wizards

An important characteristic of this team is that the implementation is *multi-threaded* [52]. This has the advantage that agents can perform various computations while waiting for the completion of slow I/O operations to and from the server. Furthermore, they have used a reinforcement learning technique called *Q-learning* (see [41]) to learn a decision-making mechanism for the agents. Although the main objective of the team is to score goals, the local goal of each individual agent is different due to the fact that they have different roles in the team. By linking these local goals together, an efficient way of cooperation emerges [51]. An additional feature of this team is that the agents possess several *Position Selection Behaviors* (PSBs) for choosing an optimal position on the field in different situations [39]. An example of such a behavior is the *Marker PSB*. This PSB selects an opponent to mark and chooses a strategic position based on the position of this opponent and the position of the ball. In the same way each agent has a *Tracker*

[3] **S**trategic **P**ositioning by **A**ttraction and **R**epulsion.

[4] The winning team of last year always participates at the next championship with an unchanged version. This team then serves as a benchmark to measure progress in the domain.

PSB, an *Offside Trap PSB*, etc. The *Essex Wizards* finished 3rd at *RoboCup-99*. One year later, they reached the same place at the *EuRoboCup-2000* European Open and became 7th at *RoboCup-2000*.

2.1.3 FC Portugal

The creators of this team decided to base their low-level implementation almost entirely on that of *CMUnited-99* and to concentrate primarily on high-level issues. One of the main innovations of the *FC Portugal* team is their positioning mechanism which is based on the distinction between *strategic* and *active* situations [76]. In strategic situations the players use a method called *Situation Based Strategic Positioning* (SBSP) to calculate a strategic position based on their player type, the current game situation and the current tactic and formation. In active situations, the player positions are calculated using specific ball possession or ball recovery mechanisms. The *FC Portugal* players also use a mechanism called *Dynamic Positioning and Role Exchange* [56], which enables them to exchange roles and positions inside their current formation. They will only do this when the utility of the exchange is positive for the team. Position exchange utilities are calculated using the distance from the player's current position to his strategic position and the importance of his position inside the formation in the current situation. Additional features of this team include their intelligent communication mechanism (ADVCOM) and their strategic looking mechanism (SLM) [56, 76]. Furthermore, the agents use a decision tree to choose an appropriate action in a given situation [77]. The *FC Portugal* high-level strategy is easily configurable and therefore flexible for different types of opponents. *FC Portugal* won at both *EuRoboCup-2000* and *RoboCup-2000*.

2.1.4 Cyberoos

The implementation of this team is based on a hierarchy of *logic-based* agent architectures which captures certain types of situated behavior[5] and some basic classes of more complex tactical behavior [70, 73, 74]. The lowest level in the hierarchy is formed by the *Clockwork Agent*, which is able to distinguish only between sensory states that have different time values and has no other sensors apart from a timer. Above that is the *Tropistic Agent* which is characterized by a broader perception-action feedback. This type of agent has different sensors, but reacts to its sensory input in a purely reactive fashion. At the next level we then find the *Hysteretic Agent*. This type of agent is defined as a reactive agent that maintains an internal state and uses this internal state together with its sensory states to activate its effectors. Above this is the *Task-Oriented Agent*, which is capable of performing certain tactical elements in real time by activating only a subset of its behavior instantiations and thus concentrating only on a specified task. Finally, the highest level is formed by the *Process-Oriented Agent*, which is capable of consolidating related tasks into coherent processes. The resulting framework has proved to be very expressive and captures several desirable properties of both the situated automata [42] and subsumption-style architectures [9], while retaining the rigour and clarity of the logic-based representation. An additional feature of this team is that they use a sophisticated agent-environment synchronization method [13] that greatly enhances the overall performance of the team. *Cyberoos* finished 3rd at the Pacific Rim Series at PRICAI-98 and became 4th at *EuRoboCup-2000*. At *RoboCup-2000* the team reached 9th place.

2.1.5 Karlsruhe Brainstormers

The main research issue that is addressed by this team is the development and application of reinforcement learning techniques in complex domains. Their long-term goal is to develop a learning system which can be

[5] A situated agent reacts to changes in the environment instead of relying on abstract representations and inferential reasoning.

given the goal 'win the match' and that can then learn to generate the appropriate behavior. However, the complexity of the robotic soccer domain (huge state space, many possible actions and strategies, partial observability of state information, etc.) makes the use of traditional reinforcement learning methods very difficult. The team tries to tackle this complexity by using *sequences* of basic commands instead of separate ones in order to reduce the number of actions and decisions available to the agent [79]. They call such sequences 'moves' and use *Real-Time Dynamic Programming* methods (see [5]) to learn these moves by incrementally approximating an optimal value function using a feedforward neural network [78]. Examples of moves which have been successfully learned are the following:

- *Kick*: kick the ball in a specified direction with the desired speed.
- *Intercept-ball*: intercept a moving ball taking the stochastic nature of the domain into account.
- *Dribble*: run with the ball without losing control of it.
- *Positioning*: move to a particular position while avoiding collisions with other players.
- *Stop-ball*: stop and control a high-speed ball.
- *Hold-ball*: keep the ball away from an opponent.

On a tactical level, the agents now have to learn which of the available moves to execute. The difficulty with this decision is that in general a complex sequence of moves has to be selected, since a single move is not likely to achieve the final goal. The current version of the *Karlsruhe Brainstormers* team uses an intermediate step to a reinforcement learning solution to this problem. They call this the *Priority−Probability−Quality* (PPQ) approach [80]. In this approach, each possible move is judged by both its usefulness (quality) and probability of success. The quality of a move is given by a priority ordering and the probability of success is learned by a simple trial-and-error training procedure. With this approach *Karlsruhe Brainstormers* has been very successful in recent competitions. The team was runner-up behind *FC Portugal* at both *EuRoboCup-2000* and *RoboCup-2000*.

2.1.6 Magma Freiburg

'Magma' stands for **m**otivation **a**ction control and **g**oal **m**anagement of **a**gents. The action control mechanism of the *Magma Freiburg* team is based on *extended behavior networks* [25]. These extend original behavior networks (see [35, 60]) to exploit information from *continuous* domains and to allow the *concurrent* execution of behaviors. Extended behavior networks consist of the following components [24, 26]:

- *Goals*. These are represented by a static importance value of the goal, a goal condition describing the situation in which the goal is satisfied and a relevance condition whose value represents the dynamic relevance of the goal. The more the current state diverges from a goal state, the more relevant (i.e. urgent) this goal becomes.
- *Competence modules*. These consist of a list of preconditions that have to be satisfied for the module to be executable, the behavior which is executed once the module is selected for execution, a list of effects expected after the behavior execution and an activation value for the module.
- *Perceptions*. In extended behavior networks these are *real-valued* propositions in order to improve the quality of perception within continuous domains.
- *Resource nodes*. These are used to coordinate the selection of multiple concurrent behaviors.

To achieve goal-directed behavior, a competence module receives activation from a goal if it has an effect that satisfies the goal. Competence modules can also be inhibited by a goal if the module has an effect preventing the goal from being satisfied. A high activation value of a module increases the probability that the corresponding behavior will be executed. Apart from the additional benefits, extended behavior networks also maintain the advantages of original behavior networks such as reactivity, planning capabilities, consideration of multiple goals and cheap computation. The approach has been successfully implemented in the *Magma Freiburg* soccer team, which was runner-up at *RoboCup-99* and reached 5th place at *RoboCup-2000*.

2.1.7 AT Humboldt

This team has a very strong low-level implementation [10]. Their agent architecture is based on the BDI (**B**elief-**D**esire-**I**ntention) approach [11]. This means that each agent consists of four different components:

- The *belief* component models the belief of the agent about the state of the environment based on sensory information.
- The *desire* component evaluates possible desires according to the beliefs.
- The *intention* component specifies the best plan according to a committed desire.
- The *execution* component receives the chosen action and is responsible for the synchronization with the *soccer server*.

In situations where agents do not have enough information to induce rules, they use *Case Based Reasoning* [57] to learn from former experiences. This requires efficient case memories to enable a quick retrieval of old cases. *AT Humboldt* won at *RoboCup-97*, was runner-up at *RoboCup-98* and became 7th at *RoboCup-99*.

2.1.8 Windmill Wanderers

This is the old team from the University of Amsterdam that was created by the late Emiel Corten. It uses a three-layer agent architecture [18]. The *Basic Layer* provides access to the functionality offered by the *soccer server* and hides the server as much as possible from the other layers. Subsystems exist for receiving and parsing information from the server and for sending actions to the server. This layer also contains a visual memory. The *Skills Layer* then uses the functionality offered by the *Basic Layer* to define several advanced agent skills. A distinction is made between essential tasks (e.g. intercept), elementary tasks (e.g. search ball) and feature extractors. A feature in this respect can be seen as a derived piece of information that is important when deciding what to do next or how to perform a certain action. Examples are 'team mate free' or 'closest to ball'. Features are calculated using the current information available in the visual memory and internal state of the agent. The highest layer in the architecture is the *Control Layer* which chooses the optimal action from the *Skills Layer* based on the current field situation. A strong aspect of the *Windmill Wanderers* team is that they use an effective zone strategy for positioning players on the field [17]. In order to improve the local positioning of players inside their zone an attraction-repulsion algorithm is used in which the attractive and repulsive forces are supplied directly by other players and indirectly by feature extractors indicating favorable locations. Furthermore, a generic learning algorithm has been implemented to find optimal parameter values for several skills such as shooting and dribbling. The *Windmill Wanderers* reached 3rd place at *RoboCup-98* and finished 9th at *RoboCup-99*[6].

[6] In this competition the team was called *UvA-Team*

2.1.9 Mainz Rolling Brains

This team uses a three-layer agent architecture [112]. At the bottom, the *Technical Layer* communicates with the server and as such it provides an abstract interface to the server for the other layers. In the middle, the *Transformation Layer* contains all the skills and tools that a player might use. Each player has the possibility to choose between two skill levels. Low-level skills correspond with basic player commands, such as 'turn' and 'kick', whereas high-level skills consist of sequences of such commands (e.g. 'intercept' or 'dribble'). The highest layer is the *Decision Layer* which can be seen as the brain of the player. The *Mainz Rolling Brains* use a hierarchical rule tree for player control. Each rule in this tree consists of a condition for firing the rule and an action[7] which is to be performed when the condition is satisfied. A significant feature of this team is that the rule trees are constructed using several AI-techniques such as *Q-Learning* and *Genetic Algorithms* [113]. Different rule trees are created for different player types and separate subtrees exist for several standard situations. The players also use models of opponent behavior constructed during a match in order to be adaptive to different strategies. The *Mainz Rolling Brains* finished 5th at *RoboCup-98* and reached the same place at *RoboCup-99*.

2.1.10 YowAI

This team has a high-quality low-level implementation. The agents use an effective synchronization scheme and have an accurate world model [106]. Furthermore, they possess very strong low-level individual skills and use a sophisticated stamina management system that cleverly controls the running behavior of the agents so that none of them actually runs too much compared with the others [107]. A significant feature of this team is that they do not use any explicit cooperation between their agents. Despite this, they have performed well at international competitions and this has led them to conclude that individual low-level skills and world model accuracy have prior importance to cooperation. Their future research goal is to investigate how cooperation can be realized without communicating detailed numerical or symbolic information such as global coordinates or elaborate plan sequences. It is their intention to demonstrate that powerful cooperation can be achieved by man-like communication in the form of short utterances. The *YowAI* team became 7th at *RoboCup-99*. In the year 2000, the team won the *Japan Open* and reached 5th place at *RoboCup-2000*.

2.1.11 Other Teams: Footux, RoboLog, Gemini

During our literature study we have also investigated a small number of other teams, some of which have been less successful in past RoboCup competitions. Here we shortly mention three of these teams that exhibited significant features.

- **Footux**. This team uses a *hybrid* agent architecture which combines the benefits of the more traditional horizontal and vertical architectures [34]. In a vertical architecture, a higher layer is called when it is needed by the layer below and it uses the functionality offered by this lower layer to perform its task. In a horizontal architecture however, the layers are more independent and all of them are active at the same time. The *Footux* architecture combines the benefits of both approaches into a hybrid architecture in which the perception flow is vertical and the action flow is horizontal.
- **RoboLog Koblenz**. This team has copied the low-level skills of *CMUnited-99* and uses multi-agent scripts implemented in the programming language *Prolog* to describe multi-agent behavior

[7] In a terminal rule the action is an elementary action; in a meta-rule the action consists of a set of subrules.

[63]. Logic is used as a control language for deciding how an agent should behave in situations where there is possibly more than one choice. The agents use logical rules in decision trees to make these choices. In order to specify the more procedural aspects of agent behavior, statecharts are adopted [64]. *Robolog Koblenz* became 5th at *EuRoboCup-2000*.

- **Gemini**. A significant feature of this team is that cooperation between agents is achieved without using inter-agent communication [65]. Reinforcement learning is used to select the best strategy against an opponent based on statistical information. This team became 7th at *RoboCup-98*, 13th at *RoboCup-99* and finished 9th at *RoboCup-2000*.

2.2 Reference Guide

We conclude this chapter by presenting an overview of references about each team that has been discussed in the previous section. Along with the references we also summarize the significant results of these teams in international competitions (top-10 finishes only). It is important to realize that the list is not exhaustive and only contains references and results up to and including the year 2000.

Team	**References**	**Roll of honour**
CMUnited	[7, 90, 92, 93, 94, 95, 96, 97] [98, 99, 100, 102, 103, 104, 115]	4th WC97, 1st WC98, 1st+9th WC99, 3rd+4th WC00
Essex Wizards	[36, 37, 39, 50, 51, 52, 53]	3rd WC99, 3rd EC00, 7th WC00
FC Portugal	[56, 76, 77]	1st EC00, 1st WC00
Cyberoos	[13, 69, 70, 71, 72, 73, 74]	3rd PR98, 4th EC00, 9th WC00
Karlsruhe Brainstormers	[79, 80]	2nd EC00, 2nd WC00
Magma Freiburg	[24, 25, 26]	2nd WC99, 5th WC00
AT Humboldt	[3, 10, 11]	1st WC97, 2nd WC98, 7th WC99
Windmill Wanderers	[17, 18]	3rd WC98, 9th WC99
Mainz Roling Brains	[111, 112, 113]	5th WC98, 5th WC99
YowAI	[106, 107]	7th WC99, 1st JO00, 5th WC00
Footux	[34]	-
RoboLog Koblenz	[63, 64, 89]	5th EC00
Gemini	[65]	7th WC98, 9th WC00

Table 2.3: References for further reading about several successful soccer simulation teams. Significant competition results of these teams are shown on the right. Here ‘WC’ denotes *World Championship*, ‘EC’ denotes *European Championship*, ‘PR’ denotes *Pacific Rim Series* and ‘JO’ denotes *Japan Open*.

Chapter 3

The RoboCup Soccer Server

The *RoboCup Soccer Server* is a soccer simulation system which enables teams of autonomous agents to play a match of soccer against each other. The system was originally developed in 1993 by Dr. Itsuki Noda (ETL, Japan). In recent years it has been used as a basis for several international competitions and research challenges. The *soccer server* provides a realistic domain in the sense that it contains many real world complexities such as sensor and actuator noise and limited perception and stamina for each agent. One of its purposes is the evaluation of multi-agent systems, in which the communication between agents is restricted. In this chapter we give a detailed description of version 7.x of the simulator. The information presented is largely based on [32] and partly the result of several experiments that we have performed when studying the behavior of the *soccer server*. We will not address every aspect of the simulation as is done in [32], but only discuss the concepts and parameters which are important for understanding the remainder of this thesis. The chapter is organized as follows. In Section 3.1 we present a general overview of the main components of the simulator. The sensor, movement and action models are discussed in Sections 3.2−3.4. Section 3.5 is devoted to the concept of heterogeneous players followed by an explanation of the referee model in Section 3.6. The use of the coach is shortly discussed in Section 3.7. The chapter is concluded in Section 3.8 which contains a summary of the most important features of the simulation.

3.1 Overview of the Simulator

The RoboCup simulator consists of three main components:

- the *soccer server*
- the *soccer monitor*
- the *logplayer*

A simulation soccer match is carried out in *client-server* style. The *soccer server* provides a domain (a virtual soccer field), simulates all the movements of objects in this domain and controls a soccer game according to several rules. The characteristics of the server are specified by a set of server parameters which will be discussed throughout this chapter. These parameters define, for example, the amount of noise that is added to visual perceptions and the maximum speed of a player. Players are controlled by client programs which act as their brain and which connect to the server through a specified port (6000). Each client program can control only a *single* player. All communication between the server and the clients is done via *UDP/IP* sockets. Using these sockets, client programs send requests to the server to perform a desired action (e.g. 'kick'). When the server receives such a message it handles the request and updates the environment accordingly. After fixed intervals the server also sends sensory information about the state of the world to each player. Although direct communication between the clients is not permitted, it is allowed for clients to communicate with each other indirectly via the server using *say* and *hear* protocols which restrict the communication. When a match is to be played, two teams each consisting of 11 separate clients make a connection with the server. The objective of each team is to direct the ball into the opponent goal, while preventing the ball from entering their own goal.

It is important to realize that the server is a real-time system working with discrete time intervals (cycles). Each cycle has a specified duration defined by the server parameter `simulator_step`[1] which in the current server version has a value of 100ms. During this period clients can send requests for player actions to the server and the server then collects these requests. It is only at the end of a cycle however, that the server executes the actions and updates the state of the world. The server thus uses a discrete action model. When a client sends multiple action requests to the server during a single cycle, the server *randomly* chooses one for execution and discards the others. It is thus important that each client sends at most one action request during a cycle. On the other hand, sending no request during a given cycle will mean that the agent misses an opportunity to act and remains idle. This is undesirable since in real-time adversarial domains this may lead to the opponents gaining an advantage. Therefore, slow decision making leading to missing action opportunities has a major impact on the performance of the team.

A complex feature of the *soccer server* is that sensing and acting are *asynchronous*. In version 7.x of the simulator, clients can send action requests to the server once every 100ms, but they only receive visual information at 150ms intervals[2]. Since it is crucial for each agent to perform an action whenever he has the opportunity, this means that in some cycles agents must act without receiving new visual information. This feature is challenging for the agents since it requires them to make a prediction about the current world state based on past perceptions. Asynchronous sensing and acting thus force agents to find an optimal balance between the need to obtain information about the world and the need to act as often as possible. Furthermore, actions that need to be executed in a given cycle must arrive at the server during the right interval. It is therefore important to have a good synchronization method for sending actions to the server, since this can greatly enhance the overall performance of the team.

The simulator also includes a visualization tool called *soccer monitor*, which allows people to see what is happening within the server during a game. The *soccer monitor* displays the virtual field from the *soccer server* on a computer screen using the X window system. The *soccer server* and *soccer monitor* are connected via *UDP/IP*. As soon as the server is connected to the monitor it will send information to the monitor each cycle concerning the current state of the world. Figure 3.1 shows the *soccer monitor* display. The information shown on the monitor includes the team names, the score, the current play mode, the current time (i.e. the number of cycles which have passed), the field boundaries and the positions of all the players and the ball. Note that each player is drawn as a two-halved circle containing a number. The light side represents the front part of the player's body, whereas the dark side is his back. The black line which is visible in the light area represents the player's neck angle and defines the direction of his vision. The number denotes the uniform number belonging to that particular player. The black bars which are visible on the left and right represent the goals. Note that the monitor also provides a visual interface to the server in the form of two buttons labeled *Kick Off* and *Quit*. When both teams have connected to the server to start a match, the *Kick Off* button allows a human referee to start the game. The *Quit* button can be used to break off the simulation, disconnecting all the clients and terminating the server.

[1] Throughout this thesis the names of server parameters are shown in typewriter font.
[2] These are the default values for the server parameters `simulator_step` and `send_step`.

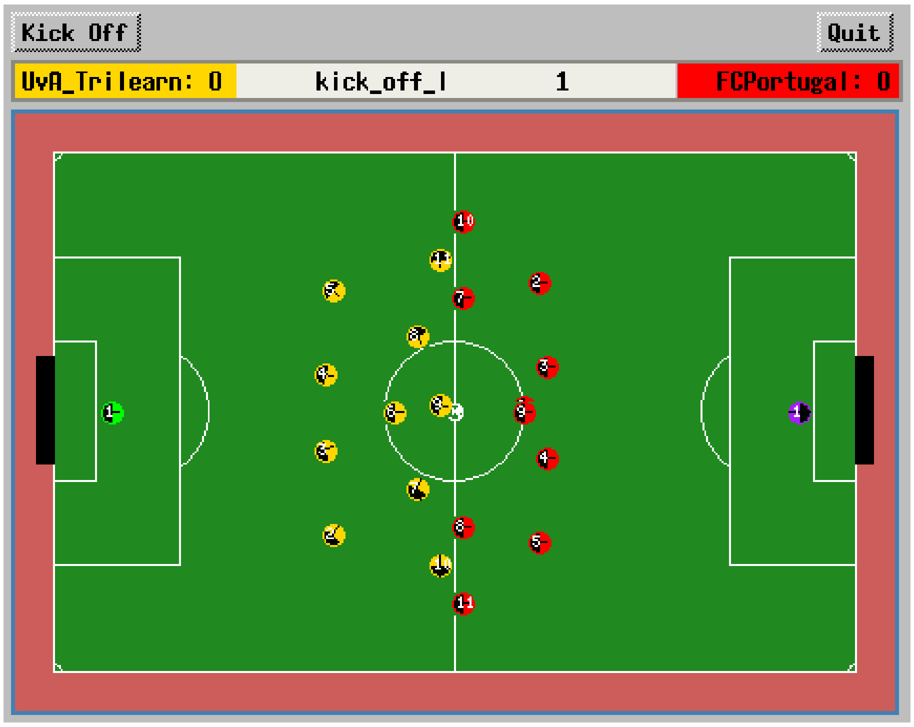


Figure 3.1: The soccer monitor display. Note that the soccer field and all objects on it are two-dimensional. The concept of 'height' thus plays no role in the simulation. The field has dimensions `pitch_length` × `pitch_width` with goals of width `goal_width`. In the current server version this means that the size of the field is 105m×68m and that the width of the goals is 14.02m. The goals are doubled in size as compared to ordinary soccer, since scoring in two dimensions is more difficult than in three.

To enforce the rules of the game, the simulator includes a referee module which controls the match. This artificial referee can detect trivial situations such as when a team scores or when the ball goes out of bounds. The referee also enforces the offside rule, controls the play mode (`kick_off`, `corner_kick`, etc.) and suspends the match when the first or second half finishes. Several situations however, such as 'obstruction' or 'ungentlemanly play', are hard to detect since the intentions of players cannot be mechanically deduced. Therefore, a human referee is used to judge this kind of fouls. The human referee can give free kicks to either team or drop the ball at a chosen spot on the field using a special server interface built into the monitor. In order to enhance their performance, teams can also make use of a coach client. The coach can be used, for example, to analyze the strengths and weaknesses of the enemy team and to give strategic advice by communicating with the players.

The third main component of the simulator is the *logplayer*. This is a tool which can be thought of as a video recorder and which can be used to replay games. During a game it is possible to run the server using an option which causes it to make a recording of the current match. This means that the server stores all the match data in a *logfile*. The *logplayer* combined with the *soccer monitor* can then be used to replay the game (i.e. the logfile) as often as needed. This can be useful for analyzing a team and for debugging clients. To facilitate fast debugging the *logplayer* is equipped with *stop*, *fast forward*, and *rewind* buttons just like a real video recorder. In addition, the *logplayer* makes it possible to jump to a particular cycle in a game. This can be useful if you only want to see a particular game situation such as a goal.

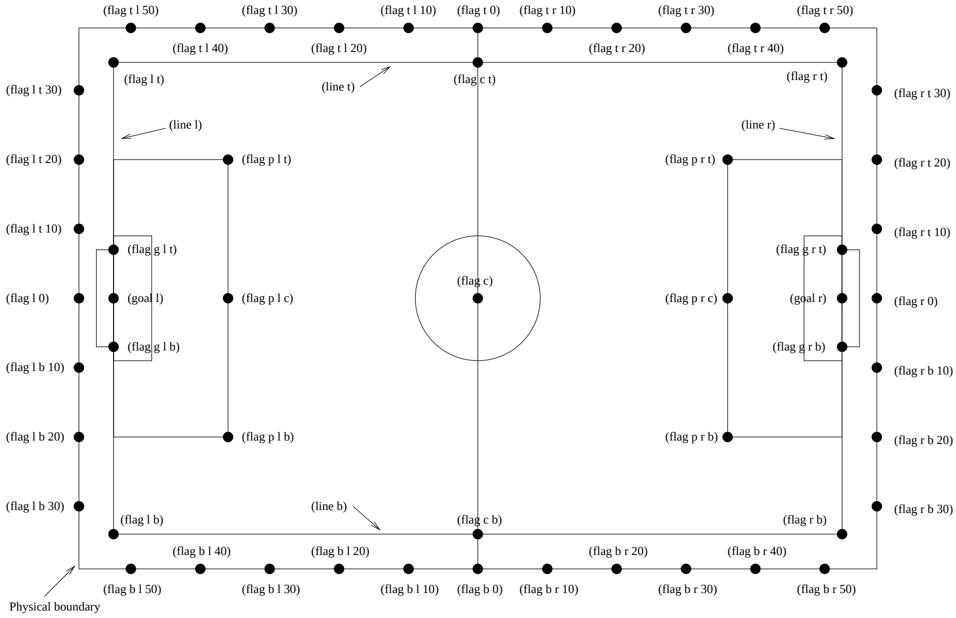


Figure 3.2: The positions and names of all the landmarks in the simulation. Taken from [32].

3.2 Sensor Models

A RoboCup agent has three different types of sensors: a *visual* sensor, a *body* sensor and an *aural* sensor. Together these sensors give the agent a reasonably good picture of its environment. In this section we discuss the characteristics of each of these three sensor types.

3.2.1 Visual Sensor Model

The visual sensor detects visual information about the field such as the distance and direction to objects in the player's current field of view. This information is automatically sent to the player every `send_step` ms. The visual sensor also works as a proximity sensor by 'seeing' objects that are close, but behind the player. It is important to realize that all visual information given is *relative* from the player's perspective. As a result a player cannot directly see his own global position or the global positions of other players and the ball. The relative information must be converted into global information however, since old relative information is of no use once the player himself has moved to another position on the field. The agents thus need a way to derive global information from a visual message. To this end, several landmarks (flags, lines and goals) have been placed on and around the field. This is illustrated in Figure 3.2 which shows the positions an names of all the landmarks in the simulation. By combining the known global positions of these landmarks with their relative positions (which are included in a visual message) an agent can determine his own global position and the global positions of the ball and other players.

A player can directly control the *frequency*, *range* and *quality* of the visual information which is sent to him. The frequency with which visual information arrives from the server is determined by the server parameter `send_step`, which represents the basic time step between visual messages and currently stands at 150ms. However, a player can choose to trade off the frequency of visual messages against the quality of the given information and the width of his view cone. He can do this by adjusting his *ViewQuality*, which can be set to either ***high*** (default) or ***low***, and *ViewWidth*, which can be set to either ***narrow***, ***normal*** (default) or ***wide***. For example, setting *ViewQuality* to ***low*** means that the player will only receive direction information to objects and no distances. However, he will receive this information twice as often. The frequency with which a player receives visual messages can be calculated as follows:

$$\textbf{\textit{view\_frequency}} = \texttt{send\_step} \cdot \textbf{\textit{view\_width\_factor}} \cdot \textbf{\textit{view\_quality\_factor}} \tag{3.1}$$

where ***view_width_factor*** is 0.5 iff *ViewWidth* is ***narrow***, 1 iff *ViewWidth* is ***normal***, and 2 iff *ViewWidth* is ***wide***; ***view_quality_factor*** is 1 iff *ViewQuality* is ***high*** and 0.5 iff *ViewQuality* is ***low***. The field of view of a player is determined by the server parameter `visible_angle`, which represents the number of degrees of a player's normal view cone, and by the player parameter *ViewWidth* (see above). In the current server version the default values for these parameters are 90 degrees and ***normal*** respectively. A player's `view_angle` is calculated according to the following equation:

$$\textbf{\textit{view\_angle}} = \textbf{\textit{visible\_angle}} \cdot \textbf{\textit{view\_width\_factor}} \tag{3.2}$$

where ***view_width_factor*** depends on *ViewWidth* as described above. Note that a player can also 'see' objects outside his view cone but within `visible_distance` (currently 3.0) meters away from him ('feel' might be more appropriate in this case). However, he will then only receive information about the type of the object (ball, player, goal or flag) and not about its name (i.e. which flag, which player, etc.).

Visual information arrives from the server in the following format:

(**see** *Time* *ObjInfo*$^{+}$)

where

Time ::= simulation cycle of the soccer server
ObjInfo ::= **(*ObjName* *Distance* *Direction*** [***DistChange* *DirChange*** [***BodyDir* *NeckDir***]]**)**
| **(*ObjName* *Direction*)**
ObjName ::= (**p** ["*Teamname*" [*UniformNr* [**goalie**]]])
| **(b)**
| **(g** [**l**|**r**]**)**
| **(f c)**
| **(f** [**l**|**c**|**r**] [**t**|**b**]**)**
| **(f p** [**l**|**r**] [**t**|**c**|**b**]**)**
| **(f g** [**l**|**r**] [**t**|**b**]**)**
| **(f** [**l**|**r**|**t**|**b**] **0)**
| **(f** [**t**|**b**] [**l**|**r**] [**10**|**20**|**30**|**40**|**50**]**)**
| **(f** [**l**|**r**] [**t**|**b**] [**10**|**20**|**30**]**)**
| **(l** [**l**|**r**|**t**|**b**]**)**
| **(B)**
| **(F)**
| **(G)**
| **(P)**
Distance ::= positive real number
Direction ::= $-180 \sim 180$ degrees
DistChange ::= real number
DirChange ::= real number
BodyDir ::= $-180 \sim 180$ degrees
NeckDir ::= $-180 \sim 180$ degrees
Teamname ::= string
UniformNr ::= $1 \sim 11$

The object information is provided for all visible objects, i.e. for all the objects in the player's view cone. The amount of information given can be different for each object depending on the type of and the distance to the object and on the quality of the visual information determined by the player parameter *ViewQuality*. When *ViewQuality* is set to ***low***, the only information given about an object is the name of the object and the direction to the object. When *ViewQuality* is ***high*** more information is provided depending on the type of the object and the distance to it. Let *dist* be the distance to the object in question. Then the situation can be summarized as follows:

- For landmarks (i.e. flags, lines and goals) the information given always consists of the name of the landmark in question, the distance to this landmark and the direction to this landmark. Note that in the case of a line, *Distance* is the distance to the point on the line where the bisector of the player's view cone crosses the line and *Direction* is the angle between the line and this bisector.
- For players **p** the amount of information given depends on *dist* in the following way:
  - If $dist \leq$ `unum_far_length`, then both the uniform number of the player and the team name of the team he belongs to are visible. Furthermore, values for *Distance*, *Direction*, *DistChange*, *DirChange*, *BodyDir* and *NeckDir* are also included in the visual message.
  - If `unum_far_length` $<$ *dist* $\leq$ `unum_too_far_length` = `team_far_length`, then the team name is always visible and values for *Distance* and *Direction* will always be included. However, the probability that the player's uniform number is visible decreases linearly from 1 to 0 as *dist* increases. The same holds for the probability that values are given for *DistChange*, *DirChange*, *BodyDir* and *NeckDir*.
  - If $dist \geq$ `unum_too_far_length` = `team_far_length`, then values for *DistChange*, *DirChange*, *BodyDir* and *NeckDir* are never included anymore, whereas information about *Distance* and *Direction* will always be given.
  - If `unum_too_far_length` = `team_far_length` $<$ *dist* $<$ `team_too_far_length`, then the uniform number is not visible and the probability that the team name is visible decreases linearly from 1 to 0 as *dist* increases. When *dist* exceeds `team_too_far_length` the team name is never visible anymore and the player is simply identified as an anonymous player.
- For the ball **b** the situation is similar:
  - If $dist \leq$ `unum_far_length`, then values for *Distance*, *Direction*, *DistChange* and *DirChange* are included in the visual message.
  - If `unum_far_length` $<$ *dist* $\leq$ `unum_too_far_length` = `team_far_length`, then values for *Distance* and *Direction* will always be included. However, the probability that *DistChange* and *DirChange* are given decreases linearly from 1 to 0 as *dist* increases.
  - If $dist \geq$ `unum_too_far_length` = `team_far_length`, then values for *DistChange* and *DirChange* are never included anymore, whereas the *Distance* and *Direction* will always be given.

In version 7 of the *soccer server*, the values for `unum_far_length`, `unum_too_far_length`, `team_far_length` and `team_too_far_length` are 20, 40, 40 and 60 meters respectively. Figure 3.3 (taken from [90]) shows the visual range of a player and provides an example of how the amount of information given about an object decreases as the distance to this object increases.

Object names always contain a letter indicating the type of the object: **p** for players, **g** for goals, **b** for the ball, **f** for flags and **l** for lines. When multiple objects of the same type exist, this letter is followed by a specifier which indicates which object of that type it concerns. We have already seen that in case of a player, the **p** is possibly followed by the team name of the player and his uniform number depending

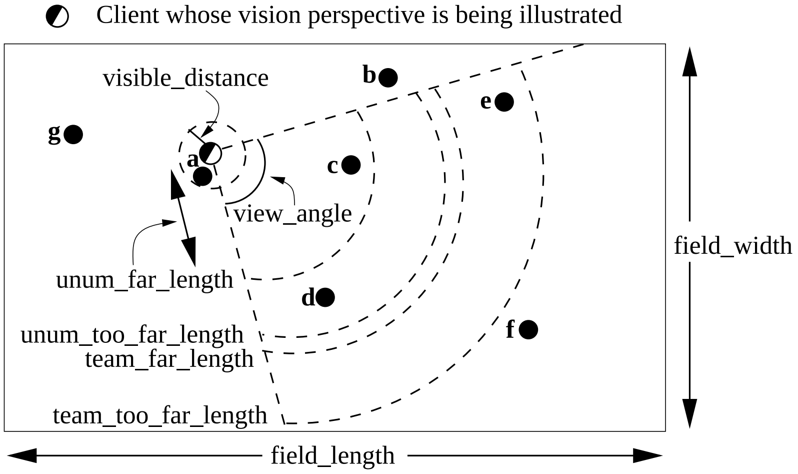


Figure 3.3: The visual range of a player. The amount of visual information about an object decreases as the distance to this object increases. In this example the sensing player is shown as a two-halved circle of which the light side is his front. The black circles represent other players. Only objects within the sensing player's `view_angle` and those within `visible_distance` of the sensing player can be seen. Players b and g are thus not visible, whereas all the others are (a can be 'felt'). Since player f is directly in front of the sensing player the reported angle to this player will be 0°; player e would be reported as being at roughly −40°, while player d is roughly at 20°. Player c will be identified by both team name and uniform number and his *Distance*, *Direction*, *DistChange*, *DirChange*, *BodyDir* and *NeckDir* will also be reported. Player d will be identified by team name and the *Distance* and *Direction* to this player will be given; furthermore there is about a 50% chance that his uniform number and values for *DistChange*, *DirChange*, *BodyDir* and *NeckDir* will be reported. For player e only the *Distance* and *Direction* will be reported for sure with about a 25% chance of getting the team name as well. Player f will simply be identified as an anonymous player for which only the *Distance* and *Direction* are given. Taken from [90].

on the distance to the player. When the player is a goalkeeper this is specified by the optional argument **goalie**. The ball is simply identified as **(b)**. The naming convention for landmarks is motivated by their position on the field. The object **(f c)**, for example, is a virtual flag at the center of the field and **(f p r t)** is a virtual flag at the top corner of the penalty area on the right hand side. Note that several types of flags are located 5 meters outside the playing area. The **(f b l 40)** flag, for example, is located 5 meters below the bottom side line and 40 meters to the left of the center line. In the same way **(f r t 20)** is located 5 meters to the right of the right side line and 20 meters above the center of the right goal. Refer back to Figure 3.2 for a good overall picture of the positions and names of all the landmarks on the field.

Values for *Distance*, *Direction*, *DistChange*, *DirChange*, *BodyDir* and *NeckDir* are calculated as follows:

$$p_{rx} = p_{xt} - p_{xo} \tag{3.3}$$
$$p_{ry} = p_{yt} - p_{yo} \tag{3.4}$$
$$v_{rx} = v_{xt} - v_{xo} \tag{3.5}$$
$$v_{ry} = v_{yt} - v_{yo} \tag{3.6}$$
$$\mathit{Distance} = \sqrt{p_{rx}^2 + p_{ry}^2} \tag{3.7}$$
$$\mathit{Direction} = \arctan(p_{ry}/p_{rx}) - a_o \tag{3.8}$$
$$e_{rx} = p_{rx}/\mathit{Distance} \tag{3.9}$$

$$
\begin{aligned}
e_{ry} &= p_{ry}/\mathit{Distance} && (3.10)\\
\mathit{DistChange} &= (v_{rx} \cdot e_{rx}) + (v_{ry} \cdot e_{ry}) && (3.11)\\
\mathit{DirChange} &= [(-(v_{rx} \cdot e_{ry}) + (v_{ry} \cdot e_{rx}))/\mathit{Distance}] \cdot (180/\pi) && (3.12)\\
\mathit{BodyDir} &= \mathit{body\_dir\_abs} - a_0 && (3.13)\\
\mathit{NeckDir} &= \mathit{neck\_dir\_abs} - a_0 && (3.14)
\end{aligned}
$$

where (p_{xt}, p_{yt}) and (v_{xt}, v_{yt}) respectively denote the global position and global velocity of the target object and (p_{xo}, p_{yo}) and (v_{xo}, v_{yo}) the global position and global velocity of the sensing player; a_0 is the global facing direction of the sensing player. Furthermore, (p_{rx}, p_{ry}) and (v_{rx}, v_{ry}) are respectively the relative position and relative velocity of the target object and (e_{rx}, e_{ry}) denotes the unit vector in the direction of the relative position. Values for *BodyDir* and *NeckDir* will only be included if the target object is a player. *BodyDir* is the body direction of the observed player relative to the neck direction of the observing player. If the body of the observed player is turned in the same direction as the neck of the observing player, the value for *BodyDir* would thus be 0. In the same way *NeckDir* is the neck direction of the observed player relative to the neck direction of the observing player.

One of the real-world complexities contained in the *soccer server* is that the precision of visual information decreases as the distance to an object increases. Noise is introduced into the visual sensor data by quantizing the values sent by the server. Distances to objects, for example, are quantized as follows:

$$
\mathit{Q\_Distance} = \mathrm{Quantize}(\exp(\mathrm{Quantize}(\ln(\mathit{Distance}), \mathit{StepValue})), 0.1) \qquad (3.15)
$$

Here *Distance* and *Q_Distance* are the *exact* and *quantized* distance values respectively and *StepValue* is a parameter denoting the quantize step. For players and the ball this parameter is equal to the server parameter `quantize_step` and for landmarks the server parameter `quantize_step_l` is used. Furthermore,

$$
\mathrm{Quantize}(V, Q) = \mathrm{rint}(V/Q) \cdot Q \qquad (3.16)
$$

where 'rint' denotes a function which rounds a value to the nearest integer. The amount of noise thus increases as the distance to the object increases. For example, when an object is roughly reported at distance 100.0 the maximum noise is about 10.0, whereas when the reported distance is roughly 10.0 the noise can be about 1.0. Values for *DistChange*, *Direction* and *DirChange* are quantized as follows:

$$
\begin{aligned}
\mathit{Q\_DistChange} &= \mathit{Q\_Distance} \cdot \mathrm{Quantize}(\mathit{DistChange}/\mathit{Distance}, 0.02) && (3.17)\\
\mathit{Q\_Direction} &= \mathrm{Quantize}(\mathit{Direction}, 1.0) && (3.18)\\
\mathit{Q\_DirChange} &= \mathrm{Quantize}(\mathit{DirChange}, 0.1) && (3.19)
\end{aligned}
$$

Here the quantize function is as shown in (3.16) and *Q_DistChange*, *Q_Direction* and *Q_DirChange* denote the quantized values for the distance change, direction and direction change respectively. Table 3.1 lists the server parameters which are important for the visual sensor model together with their default values.

Parameter	**Value**	**Parameter**	**Value**
`send_step`	150	`team_far_length`	40.0
`visible_angle`	90.0	`team_too_far_length`	60.0
`visible_distance`	3.0	`quantize_step`	0.1
`unum_far_length`	20.0	`quantize_step_l`	0.01
`unum_too_far_length`	40.0		

Table 3.1: Server parameters which are important for the visual sensor model with their default values.

3.2.2 Aural Sensor Model

The aural sensor detects spoken messages which are sent when a player or a coach issues a **say** command. Calls from the referee are also received as aural messages (possible referee messages are discussed later in Section 3.6). The *soccer server* communication paradigm models a crowded, low-bandwidth environment in which the agents from both teams use a single, unreliable communication channel [90]. Spoken messages are *immediately* broadcast to all nearby players from both teams without perceptual delay. Aural sensor messages arrive from the server in the following format:

(**hear** *Time Sender* "*Message*")

where

- *Time* indicates the current simulation cycle of the soccer server.
- *Sender* can be one of the following:
  - **online_coach_left** or **online_coach_right** when the sender is one of the online coaches.
  - **referee** when the sender is the referee.
  - **self** when you are the sender yourself.
  - the relative direction to the sender if the sender is another player.
- *Message* is a string representing the contents of the message; the length of the string is limited to `say_msg_size` (currently 512) bytes.

Note that there is no information about which player has sent the message or about the distance to the sender. Furthermore, the capacity of the aural sensor is limited. The server parameter `hear_max` represents the maximum hearing capacity of a player. Each time when a player hears a message his hearing capacity is decreased by `hear_decay`. Every cycle the hearing capacity of a player is then increased by `hear_inc` until it reaches `hear_max`. Since the hearing capacity of a player cannot become negative, a player can only hear a message if his hearing capacity is at least `hear_decay`. With the current server parameter values this means that a player can hear at most one message every second simulation cycle. When multiple messages arrive during this time, the first one is chosen according to their order of arrival and the rest are discarded[3]. The communication is thus extremely unreliable. However, messages from the referee are treated as privileged and are always transmitted to all the players. Since all 22 players on the field use the same communication channel, it would be possible to make the communication of the opponent team useless by overloading the channel with messages of your own. To avoid this, the players have separate hearing capacities for each team. This means that a player can hear a single message from each team every two simulation cycles. Besides this, players also have to deal with the fact that their communication range is limited. A spoken message is transmitted only to players within `audio_cut_dist` meters from the speaker. Messages from the referee however can be heard by all the players. The server parameters which are important for the aural sensor model are listed in Table 3.2 together with their default values.

Parameter	Value	Parameter	Value
`say_msg_size`	512	`hear_decay`	2
`hear_max`	2	`audio_cut_dist`	50.0
`hear_inc`	1		

Table 3.2: Server parameters which are important for the aural sensor model with their default values.

[3]This does not hold for your own messages, i.e. a player can **say** a message and **hear** one from another player simultaneously.

3.2.3 Body Sensor Model

The body sensor reports physical information about the player, such as its stamina, speed and neck angle. This information is automatically sent to the player every `sense_body_step` (in the current server version 100) ms. Body information arrives from the server in the following basic format:

(**sense_body** *Time*
(**view_mode** *ViewQuality ViewWidth*)
(**stamina** *Stamina Effort*)
(**speed** *AmountOfSpeed DirectionOfSpeed*)
(**neck_angle** *NeckAngle*)
(**kick** *KickCount*)
(**dash** *DashCount*)
(**turn** *TurnCount*)
(**say** *SayCount*)
(**turn_neck** *TurnNeckCount*)
(**catch** *CatchCount*)
(**move** *MoveCount*)
(**change_view** *ChangeViewCount*))

where

- *Time* indicates the current simulation cycle of the soccer server.
- *ViewQuality* is either ***high*** or ***low*** and denotes the quality of the visual information received.
- *ViewWidth* is either ***narrow***, ***normal*** or ***wide*** and denotes the width of the player's view cone.
- *Stamina* is a positive real number representing the player's current stamina.
- *Effort* is a positive real number representing the player's current effort capacity.
- *AmountOfSpeed* represents the player's current speed. Noise is incorporated into this value by quantizing (see Equation 3.16) the speed as follows: *AmountOfSpeed* = Quantize(*Speed*,0.01).
- *DirectionOfSpeed* represents the direction of the player's current speed. Noise is incorporated into this value by quantizing the direction according to Equation 3.18.
- *NeckAngle* represents the direction of the player's neck relative to his body (quantized as in (3.18)).
- the *Count* variables are counters for the total number of commands of a certain type which have been executed by the server; e.g. when *KickCount* = 45 this means that the player has so far executed 45 **kick** commands. These counters can be used to check whether a command sent in the previous cycle has been performed (if so then the corresponding counter value has been incremented).

The parameters mentioned here will be described more extensively in the sections where they are actually used. Server parameters that affect the body sensor are listed in Table 3.3 along with their default values.

Parameter	**Value**
`sense_body_step`	100

Table 3.3: Server parameters which are important for the body sensor model with their default values.

3.3 Movement Model

In the *soccer server* object movement is simulated stepwise in a simple way: the velocity of an object is added to its position, while the velocity decays by a certain rate and increases by the acceleration of the object resulting from certain client commands. During each simulation cycle the movement of mobile objects (i.e. players and the ball) is calculated according to the following equations:

$$\begin{aligned}
(u_x^{t+1}, u_y^{t+1}) &= (v_x^t, v_y^t) + (a_x^t, a_y^t) + (\tilde{r}_1, \tilde{r}_2) + (w_1, w_2)\text{: accelerate} && (3.20)\\
(p_x^{t+1}, p_y^{t+1}) &= (p_x^t, p_y^t) + (u_x^{t+1}, u_y^{t+1})\text{: move} && (3.21)\\
(v_x^{t+1}, v_y^{t+1}) &= \mathit{Decay} \times (u_x^{t+1}, u_y^{t+1})\text{: decay speed} && (3.22)\\
(a_x^{t+1}, a_y^{t+1}) &= (0, 0)\text{: reset acceleration} && (3.23)
\end{aligned}$$

Here (p_x^t, p_y^t), (v_x^t, v_y^t), (a_x^t, a_y^t) respectively denote the position, velocity and acceleration of the object in cycle t with $(\tilde{r}_1, \tilde{r}_2)$ and (w_1, w_2) a noise vector and a wind vector which are added to the object movement. *Decay* is a parameter representing the velocity decay rate of the object; for players this is equal to the server parameter `player_decay` and for the ball the server parameter `ball_decay` is used. The acceleration of an object results from certain client commands which are sent to the server. The acceleration of a player, for example, is a result of the player dashing. In the same way, the acceleration of the ball results from a player kicking it. These and other action commands will be described in Section 3.4. If two objects overlap at the end of a cycle, i.e. two objects collide with each other after a movement, then these objects are moved back into the direction where they came from until they do not overlap anymore; after this their velocities are multiplied by -0.1. Note that it is thus possible for the ball to go through a player as long as the ball and the player never overlap at the end of a simulation cycle.

In order to reflect unexpected movements of objects in the real world, the *soccer server* adds uniformly distributed random noise to the movement of all objects. The noise vector in Equation 3.20 contains two random numbers $\tilde{r}_i$ as its elements which are taken from a uniform distribution over the range $[-r_{max}, r_{max}]$. The value of r_{max} depends on the speed of the object and is calculated as follows:

$$r_{max} = \mathit{Rand} \cdot \|(v_x^t, v_y^t) + (a_x^t, a_y^t)\| \qquad (3.24)$$

Here *Rand* is a parameter representing the random error in the object movement; for players this is equal to the server parameter `player_rand` and for the ball the server parameter `ball_rand` is used.

Besides this, the *soccer server* also models a wind vector (w_1, w_2) as a more natural form of noise. The wind in the simulation is represented by the vector $(w_x, w_y) = \pi(\texttt{wind\_force}, \texttt{wind\_dir})$ where π is a method that converts polar coordinates to Cartesian coordinates. The actual vector (w_1, w_2) which is added to the movement of an object as a result of this wind depends on the weight of the object and is calculated according to the following formula:

$$(w_1, w_2) = \|(v_x^t, v_y^t) + (a_x^t, a_y^t) + (\tilde{r}_1, \tilde{r}_2)\| \cdot \frac{(w_x + \tilde{e}_1, w_y + \tilde{e}_2)}{\mathit{Weight} \cdot 10000} \qquad (3.25)$$

where $\tilde{e}_i$ is a random number taken from a uniform distribution over the range $[-\texttt{wind\_rand}, \texttt{wind\_rand}]$. Furthermore, *Weight* is a parameter representing the weight of the object; for players this is equal to the server parameter `player_weight` and for the ball it equals `ball_weight`. It must be noted that in the current server version ($7.x$) the values for all wind-related parameters have been set to 0.0, i.e. no wind is used. Table 3.4 shows the server parameters which are important for the movement model of *soccer server* together with their default values.

Parameter	Value	Parameter	Value
`ball_decay`	0.94	`player_weight`	60.0
`ball_rand`	0.05	`wind_force`	0.0
`ball_weight`	0.2	`wind_dir`	0.0
`player_decay`	0.4	`wind_rand`	0.0
`player_rand`	0.1		

Table 3.4: Server parameters which are important for the movement model with their default values.

3.4 Action Models

When a player client wants to perform a certain action he sends an action command to the server, thereby requesting the server to execute the action that he desires. A player can perform the following actions: **kick**, **dash**, **turn**, **say**, **turn_neck**, **catch**, **move**, **change_view**, **sense_body** and **score**. In *soccer server* version 7.*x*, the **sense_body** command has become obsolete, since body information is automatically provided every cycle by the body sensor. In earlier versions this was not the case and a player could specifically ask the server for his physical status using this command. When a **sense_body** command is given in a certain cycle, the server will return the same body information reported by the body sensor in that cycle (see Section 3.2.3 for the format of this message). When a player issues a **score** command, the server returns a message of the form:

(**score** *Time OurScore TheirScore*)

where *Time* indicates the current simulation cycle, *OurScore* is an integer denoting the number of goals scored by the player's own team and *TheirScore* is an integer denoting the number of goals scored by the opponent team. The other actions listed above all change the *state* of the world in some way. In the remainder of this section we will separately discuss the characteristics of each of these actions.

3.4.1 Kick Model

When a player wants to kick the ball, he must send a **kick** command to the server. This command takes two parameters: the *Power* of the kick and the *Angle* towards which the ball is kicked relative to the body of the kicking player. The kick power determines the amount of acceleration given to the ball and must be between `minpower` and `maxpower`; the kicking angle is given in degrees and must be between `minmoment` and `maxmoment`. When a **kick** command arrives at the server, the command will only be executed if the ball is kickable for the player that issued the command. A player can **kick** the ball if it is within the maximal kick distance defined as `ball_size` + `player_size` + `kickable_margin`. In other words, this means that a **kick** can be performed when the distance between the center of the ball and the center of the player minus the radius of the ball minus the radius of the player is between 0 and `kickable_margin`.

An important aspect of the kick model, is that the actual power with which the ball is kicked is not equal to the *Power* argument supplied to the **kick** command, but depends on the relative position (angle and distance) of the ball to the kicking player. For example, when the angle of the ball relative to the body of the player is 0° this has no negative influence on the kick power. The larger this angle gets however, the more the actual power will be reduced. In the worst case, the ball will be at a 180° angle to the player (i.e. behind him) and the actual power is then reduced by 25%. For the distance the situation is similar. When the minimum distance between the outer shape of the player and the ball equals 0 meters, this again leaves the kick power unaltered. However, when in the worst case this distance equals `kickable_margin`, the actual kick power will be reduced by 25%. If in this case the ball is also behind the player, the *Power* argument supplied to the **kick** command will thus be reduced by a total of 50%. The actual power *act_pow* with which the ball is kicked is determined according to the following formula:

$$act\_pow \quad = \quad Power \cdot \left(1 - 0.25 \cdot \frac{dir\_diff}{180} - 0.25 \cdot \frac{dist\_diff}{\texttt{kickable\_margin}}\right) \tag{3.26}$$

where *Power* is the kick power supplied as the first argument to the **kick** command, *dir_diff* is the absolute angle between the ball and the player's body direction and *dist_diff* is the distance between the player and the ball (i.e. the minimum distance between the outer shape of the player and the outer shape of the ball). The actual kick power *act_pow* is used to calculate an acceleration vector $\vec{a}_t$ that will be added to the global ball velocity $\vec{v}_t$ during cycle t as shown by Equation 3.20. This acceleration $\vec{a}_t$ is applied at the transition from simulation cycle t to $t+1$. The vector $\vec{a}_t$ is calculated as follows:

$$(a_x^t, a_y^t) \quad = \quad act\_pow \times \texttt{kick\_power\_rate} \times (\cos(\theta^t), \sin(\theta^t)) + (\tilde{k}_1, \tilde{k}_2) \tag{3.27}$$

where `kick_power_rate` is a server parameter which is used to determine the size of the acceleration vector and θ^t is the direction in which the ball is accelerated in cycle t. This direction equals the sum of the body direction of the kicking player and the *Angle* parameter of the **kick** command. Furthermore, noise is added in the form of a small vector $(\tilde{k}_1, \tilde{k}_2)$ with $\tilde{k}_i$ a random number taken from a uniform distribution over the range $[-k_{max}, k_{max}]$. Here the value of k_{max} depends on the power supplied to the **kick** command and is calculated according to the following formula:

$$k_{max} \quad = \quad \texttt{kick\_rand} \cdot \frac{Power}{\texttt{max\_power}} \tag{3.28}$$

with `kick_rand` a server parameter[4]. The acceleration vector $\vec{a}_t$ is then normalized to a maximum length of `ball_accel_max`. In the current server version the value for `ball_accel_max` is such that it equals the product of `maxpower` and `kick_power_rate`. This means that it is possible to obtain the maximum acceleration when the ball is directly in front of the kicking player and is kicked with maximum power. As indicated in Equation 3.20, the acceleration vector $\vec{a}_t$ is then added to the current velocity $\vec{v}_t$ of the ball. The resulting velocity vector is normalized to a maximum length of `ball_speed_max` and it is only after this normalization step that noise and wind are added to the ball velocity. In *soccer server* version $7.x$, the values for `ball_accel_max` and `ball_speed_max` are such that it is possible to give the maximum speed to the ball using a single kick. In earlier server versions this was not possible and multiple kicks in successive cycles were needed to make the ball go faster[5]. Note that since the new ball position is calculated as a vector addition (see Equation 3.21), the maximum distance that the ball can travel between two simulation cycles is equal to `ball_speed_max` when noise and wind are neglected. Also note that the ball does not get accelerated by other means than kicking, i.e. after the kick the acceleration of the ball remains 0 (see Equation 3.23) until another kick is performed. With the current server settings, the ball can travel a distance of about 45 meters assuming an optimal kick. In this case the ball will have covered a distance of about 28 meters after 15 cycles with a remaining velocity of about 1 meter per cycle. After 53 cycles the remaining velocity becomes smaller than 0.1 meters per cycle and the distance covered is about 43 meters. Table 3.5 shows the server parameters which are important for the kick model of *soccer server* together with their default values.

[4] With the current server settings the value for `kick_rand` equals 0.0 for default players. In the sequel we will therefore assume that no noise is added. For heterogeneous player types however, this is not the case as will be discussed in Section 3.5.

[5] In order to give maximum speed to the ball on a kick, players would first kick the ball around themselves several times in successive cycles, thereby continually increasing its velocity. The final kick would then be executed once it was possible to give maximum speed to the ball. This principle was first introduced by the *RoboCup-97* champion *AT Humboldt* (see [10]).

Parameter	Value	Parameter	Value
minpower	-100	kickable_margin	0.7
maxpower	100	kick_power_rate	0.027
minmoment	-180	kick_rand	0.0
maxmoment	180	ball_accel_max	2.7
ball_size	0.085	ball_speed_max	2.7
player_size	0.3		

Table 3.5: Server parameters which are important for the kick model together with their default values.

3.4.2 Dash and Stamina Model

The **dash** command can be used by a player to accelerate himself in the direction of his body. This command takes a single parameter: the *Power* of the dash. The dash power determines the amount of acceleration of the player and must be between `minpower` and `maxpower`. When the *Power* argument in the **dash** command is positive, the player is accelerated in forward direction; if *Power* is negative the player dashes backwards. The *soccer server* prevents players from constantly running at maximum speed (`player_speed_max`) by assigning a limited stamina to each of them. At the beginning of a half, the stamina of each player is set to the value of the server parameter `stamina_max`. Every time when a player performs a **dash**, a certain amount of his stamina is consumed. Dashing backwards is more expensive than dashing forward: on a forward dash ($Power > 0$) a player's stamina is reduced by the *Power* of the **dash** command, whereas on a backward dash ($Power < 0$) his stamina is reduced by $2 \cdot |Power|$. When a player's stamina is lower than the amount needed for a dash, the *Power* argument of the dash is reduced in such a way that it does not require more than the available stamina.

A player's stamina is modeled in three parts:

- *Stamina*: represents the current stamina of a player which must have a value somewhere between 0 and `stamina_max`; the *Power* parameter of a **dash** command cannot exceed this value.
- *Effort*: represents the efficiency of player movement and lies between `effort_min` and `effort_max`.
- *Recovery*: influences the rate at which stamina is restored; it lies between `recover_min` and 1.0.

A player's stamina is decreased when he dashes and gets restored slightly in each cycle. Every cycle in which the value for *Stamina* lies below a certain threshold, the values for *Effort* and *Recovery* are decreased until a minimum is reached[6]. If the value for *Stamina* lies above another threshold then *Effort* is increased up to a maximum, whereas the *Recovery* value is never increased[7]. Algorithm 3.1 shows the stamina model algorithm which is applied in each simulation step in more detail.

As in the kick model (see Section 3.4.1), the *actual* power with which a player dashes is not necessarily equal to the *Power* argument in the **dash** command. When a **dash** command arrives at the server, the stamina of the player that issued the command is first reduced by the *Power* argument of the dash (or by twice the absolute *Power* in case of a backward dash). The server then calculates the actual dash power *act_pow*, which in this case depends on the current *Effort* of the player in the following way:

$$act\_pow = Effort \cdot Power \tag{3.29}$$

[6] As of *soccer server* version 7.08 it is possible to see on the *soccer monitor* display when players become tired. In this case the color of a player's back fades and becomes whiter as his stamina drops. Color returns slowly when stamina is restored.
[7] It is restored to 1.0 at the beginning of each half.

```
// reduce stamina according to dash power
if Power < 0 then
  Stamina = Stamina − 2 · |Power|
else
  Stamina = Stamina − Power
end if

// reduce recovery when stamina is below recovery decrement threshold
if Stamina ≤ recover_dec_thr · stamina_max then
  if Recovery > recover_min then
    Recovery = Recovery − recover_dec
  end if
  Recovery = Max(recover_min, Recovery)
end if

// reduce effort when stamina is below effort decrement threshold
if Stamina ≤ effort_dec_thr · stamina_max then
  if Effort > effort_min then
    Effort = Effort − effort_dec
  end if
  Effort = Max(effort_min, Effort)
end if

// increase effort when stamina is above effort increment threshold
if Stamina ≥ effort_inc_thr · stamina_max then
  if Effort < effort_max then
    Effort = Effort + effort_inc
  end if
  Effort = Min(effort_max, Effort)
end if

// restore stamina according to recovery value
Stamina = Min(stamina_max, Stamina + Recovery · stamina_inc_max)
```

Algorithm 3.1: The stamina model algorithm which is applied in each simulation step.

In the current server version the maximum effort value for a player is 1.0 (=`effort_max`). As long as a player manages his stamina in such a way that his *Effort* value never decreases, the actual dash power will thus always be equal to the *Power* argument supplied to the **dash** command. The actual dash power *act_pow* is used to generate an acceleration vector $\vec{a}_t$ for the player in cycle t. This acceleration $\vec{a}_t$ is applied at the transition from simulation cycle t to simulation cycle $t+1$ and is calculated as follows:

$$(a_x^t, a_y^t) \quad = \quad act\_pow \times \texttt{dash\_power\_rate} \times (\cos(\theta^t), \sin(\theta^t)) \tag{3.30}$$

where `dash_power_rate` is a server parameter which is used to determine the size of the acceleration vector and θ^t is the body direction of the player in cycle t. The acceleration vector $\vec{a}_t$ is normalized to a maximum length of `player_accel_max`, after which it is added to the current velocity $\vec{v}_t$ of the player as shown by Equation 3.20. The resulting velocity vector is then normalized to a maximum length of `player_speed_max` and since the new player position is calculated as a vector addition (see Equation 3.21) this thus equals the maximum distance that a player can cover between two simulation cycles (neglecting noise and wind which are added after the normalization). Note that a single **dash** will only set the acceleration vector for one simulation cycle, after which the velocity of the player will decay. The acceleration of the player will then remain 0 until he performs another **dash**. In order to keep up a sustained run over time, the player must thus keep sending **dash** commands to the server. Table 3.6 shows the server parameters which are important for the dash and stamina models of *soccer server* together with their default values.

Parameter	**Value**	**Parameter**	**Value**
`minpower`	-100	`effort_inc_thr`	0.6
`maxpower`	100	`effort_inc`	0.01
`stamina_max`	4000.0	`recover_dec_thr`	0.3
`stamina_inc_max`	45.0	`recover_dec`	0.002
`effort_min`	0.6	`recover_min`	0.5
`effort_max`	1.0	`player_accel_max`	1.0
`effort_dec_thr`	0.3	`player_speed_max`	1.0
`effort_dec`	0.005	`dash_power_rate`	0.006

Table 3.6: Server parameters which are important for the dash and stamina models with default values.

3.4.3 Turn Model

When a player wants to turn (i.e. change his body direction) he must send a **turn** command to the server. This command takes the angle (also called *Moment*) of the turn as its only parameter which has valid values between `minmoment` and `maxmoment` degrees. As in the previous action models, the *actual* angle by which a player turns does not always equal the *Moment* argument in the **turn** command. Instead, it depends on the speed of the player. When a player has zero velocity, the angle that he will turn is equal to the *Moment* argument of the **turn** command with noise added. As the player moves faster however, it is more difficult for him to turn as a result of his inertia. The actual angle *act_ang* that a player turns when he issues a **turn** command is calculated as follows:

$$act\_ang = \frac{(1.0 + \tilde{r}) \cdot Moment}{1.0 + \texttt{inertia\_moment} \cdot player\_speed} \tag{3.31}$$

where $\tilde{r}$ is a random number taken from a uniform distribution over the $[-\texttt{player\_rand}, \texttt{player\_rand}]$ interval, *Moment* is the argument supplied to the **turn** command, `inertia_moment` is a server parameter denoting the inertia of a player and *player_speed* is the current speed of the turning player. In the current server version the values for `inertia_moment`, `minmoment` and `maxmoment` are 5.0, -180 and 180 respectively for default players. When a default player moves at his maximum speed of 1.0 (`player_speed_max`), the maximum *effective* turn that he can do is thus ±30 degrees (assuming no noise). However, since a player cannot **dash** and **turn** in the same cycle, the maximum speed of a player when executing a **turn** equals $\texttt{player\_speed\_max} \cdot \texttt{player\_decay} = 1.0 \cdot 0.4 = 0.4$; in this case the *effective* turn is ±60 degrees. Table 3.7 shows the server parameters which are important for the turn model together with their default values.

Parameter	**Value**	**Parameter**	**Value**
`minmoment`	-180	`player_rand`	0.1
`maxmoment`	180	`inertia_moment`	5.0

Table 3.7: Server parameters which are important for the turn model together with their default values.

3.4.4 Say Model

When a player wants to broadcast a message to other players, he can use the **say** command. This command takes a single parameter: the *Message* that the player wants to communicate. The length of the message is restricted to `say_msg_size` (currently 512) characters, which must each come from the set `[0..9a..zA..Z().+-*/?_<>]` (without the square brackets). Spoken messages are *immediately* broadcast to players from both teams[8] without perceptual delay. However, each player has a limited communication range: a spoken message is transmitted only to players within `audio_cut_dist` meters from the speaker. Furthermore, players have a limited hearing capacity. This is modeled by the server parameters `hear_max`, `hear_inc` and `hear_decay` (refer back to Section 3.2.2 for a detailed explanation of the meaning of these parameters). With the current server settings, each player can only hear one message from a teammate every two simulation cycles. This means that although a player can speak as often as he wants (even multiple times during the same cycle), it is useless to speak more frequently than that. Table 3.8 shows the server parameters which are important for the say model together with their default values.

Parameter	Value	Parameter	Value
say_msg_size	512	hear_inc	1
audio_cut_dist	50.0	hear_decay	2
hear_max	2		

Table 3.8: Server parameters which are important for the say model together with their default values.

3.4.5 Turn Neck Model

A player can turn his neck somewhat independently of his body using a **turn_neck** command. This command takes the *Angle* of the turn as its only parameter which has valid values between `minneckmoment` and `maxneckmoment` degrees. By turning his neck a player changes the angle of his head relative to his body and as a result his field of view changes. Note that a player's field of view also changes when he issues a **turn** command even if no **turn_neck** command is executed. This is because the neck angle of a player is relative to his body and the body angle changes as a result of the **turn**. As in the real world, a player cannot turn his neck indefinitely in the same direction. The minimum and maximum angle of a player's neck relative to his body are given by the server parameters `minneckang` and `maxneckang`. When the neck angle would assume an illegal value as a result of a **turn_neck** command, the *Angle* argument of the command is adapted in such a way that the neck angle remains within the allowed boundaries. It is important to realize that a **turn_neck** command can be executed in the same cycle as a **kick**, **dash** or **turn** command. Furthermore, the actual angle by which a player turns his neck is always equal to the *Angle* argument of the **turn_neck** command, i.e. no noise is added and **turn_neck** is not affected by momentum like **turn** is. Table 3.9 shows the server parameters which are important for the turn neck model together with their default values.

Parameter	Value	Parameter	Value
minneckmoment	-180	minneckang	-90
maxneckmoment	180	maxneckang	90

Table 3.9: Server parameters which are important for the turn neck model with their default values.

[8] Recall from Section 3.2.2 that both teams use the same communication channel.

3.4.6 Catch Model

When a goalkeeper wants to catch the ball he can do this by sending a **catch** command to the server. This command takes the *Direction* in which he catches the ball as its only parameter which has valid values between `minmoment` and `maxmoment` degrees. When a **catch** command arrives at the server, it will only be executed if several preconditions are met. First of all, the goalkeeper is the only player that can perform a catch. He can do this in play mode `play_on` in any direction as long as the ball is inside his own penalty box and inside his catchable area. If these conditions are not satisfied, the catch will fail. A goalkeeper's catchable area is defined as a rectangle with length `catchable_area_l` and width `catchable_area_w` in the direction of the catch. This is illustrated in Figure 3.4 which shows the catchable area of a goalkeeper when performing a catch at a $-45°$ angle. If the ball is inside the catchable area, the goalkeeper will catch the ball with probability `catch_probability`. When a catch fails, the goalkeeper cannot issue another **catch** command until `catch_ban_cycle` simulation cycles have passed; **catch** commands issued during this time have no effect. When the goalkeeper does succeed in catching the ball, the play mode will change to `goalie_catch_ball_x` and then to `free_kick_x` where `x` is either `l` or `r` denoting the left or right team. The goalkeeper can then use the **move** command (see Section 3.4.7) to move with the ball inside his own penalty area. He can do this `goalie_max_moves` times before he kicks the ball. The server parameters which are important for the catch model are shown in Table 3.10 together with their default values.

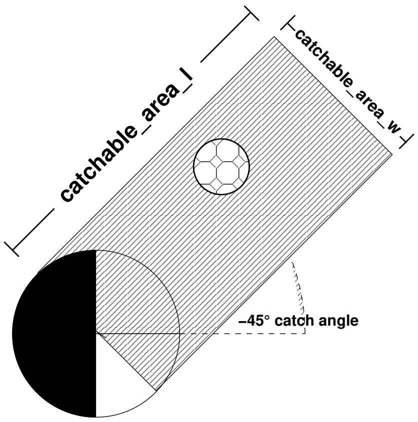


Figure 3.4: The catchable area of a goalkeeper when performing a catch at a $-45°$ angle. From [32].

Parameter	Value	Parameter	Value
`minmoment`	-180	`catch_probability`	1.0
`maxmoment`	180	`catch_ban_cycle`	5
`catchable_area_l`	2.0	`goalie_max_moves`	2
`catchable_area_w`	1.0		

Table 3.10: Server parameters which are important for the catch model with their default values.

3.4.7 Move Model

The **move** command can be used to place a player *directly* onto a desired position on the field. The command takes two parameters, X and Y, denoting the x- and y-coordinate of this position. X must have a value between $-$`pitch_length`/2 and `pitch_length`/2; Y must be between $-$`pitch_width`/2 and pitch_width/2[9]. The **move** command cannot be used during normal play (i.e. when the play mode is play_on), but exists for two purposes only. First of all, the command is used to set up the team formation at the beginning of a half (when the play mode equals before_kick_off) or after a goal has been scored (play mode is goal_l_n or goal_r_n). In these situations, a player can be placed onto any position on his own half of the field and can be moved any number of times as long as the play mode does not change. When a player is moved to a position on the opponent half, the server moves him back to a *random* position on his own half. Secondly, the **move** command is used by the goalkeeper to move inside his own penalty area after catching the ball (see Section 3.4.6). The goalkeeper is allowed to move with the ball goalie_max_moves times before kicking it. Additional **move** commands have no effect. Table 3.11 shows the server parameters which are important for the move model together with their default values.

Parameter	Value	Parameter	Value
pitch_length	52.5	goalie_max_moves	2
pitch_width	34.0		

Table 3.11: Server parameters which are important for the move model together with their default values.

3.4.8 Change View Model

Using the **change_view** command, a player can directly control the *range* and *quality* of the visual information which is sent to him by the server. This command takes two parameters: the *Width* of the player's view cone and the *Quality* of the visual information. Valid values for the *Width* parameter are ***narrow***, ***normal*** or ***wide***; *Quality* must be either ***high*** or ***low***. A **change_view** command can be issued multiple times during the same cycle. A player can use the **change_view** command to trade off the width of his view cone and the quality of the visual information against the frequency with which visual information arrives from the server. A higher view quality and a wider view cone, for example, will lead to less frequent visual information. Refer back to Section 3.2.1 for a more detailed discussion.

3.4.9 Actions Overview

The actions which have been described in this section can be divided into two distinct categories: *primary* actions (**kick**, **dash**, **turn**, **catch** and **move**) and *concurrent* actions (**say**, **turn_neck**, **change_view**, **sense_body** and **score**). During each cycle only one primary action can be executed, whereas multiple concurrent actions can be performed simultaneously with any primary action. If an agent sends more than a single primary action command during a cycle, the server randomly chooses one for execution and discards the others. Table 3.12 summarizes all the actions which can be performed by an agent and shows when and how often the commands can be executed by the server.

3.5 Heterogeneous Players

A new feature of *soccer server* version 7 was the introduction of *heterogeneous* players. In earlier server versions all the players on the field were physically identical and the values for the player parameters were the same for each player. In *soccer server* version 7 however, each team can choose from several different player types with different characteristics. These player types are randomly generated when the server is

[9] When specifying positions in global coordinates, the coordinate system is such that the negative x-direction for a team is towards the goal it defends; the negative y-direction is towards the left side of the field when facing the opponent's goal.

Syntax	**Arguments**	**Executed**	**Frequency limit**
(**kick** double double)	power, direction	end of cycle	one of these per cycle
(**dash** double)	power	end of cycle	
(**turn** double)	moment	end of cycle	
(**move** double double)	(x,y) position	end of cycle	
(**catch** double)	angle	end of cycle	
(**say** str)	message string	instantly	none (max. one heard every two cycles)
(**turn_neck** double)	moment	end of cycle	one per cycle
(**change_view** str str)	width, quality	instantly	one per cycle
(**score**)	-	instantly	none
(**sense_body**)	-	instantly	three per cycle

Table 3.12: Overview of all action commands which are available to *soccer server* agents.

started. In a match, both teams choose their players from the same set of types. The number of player types generated equals the value of the server parameter `player_types` (currently 7). Out of these types, type 0 is the default player and is always the same[10], whereas the other types are different each time when the server is started. The non-default players all have different abilities based on certain trade-offs which are defined by the values of several heterogeneous player parameters. An example is that these players are usually faster than the default player, but also get tired more quickly. When a client program connects to the server, the server sends it a number of messages[11] some of which contain information about the generated player types. For each available type, the server sends a message with the following format:

(player_type ***id player_speed_max stamina_inc_max player_decay inertia_moment dash_power_rate player_size kickable_margin kick_rand extra_stamina effort_max effort_min*****)**

These messages thus define the abilities of each player type by providing specific values for several player-related parameters which hold for that type. For the default player type the values of these parameters are fixed and for the other types they are randomly chosen from different intervals which are defined by certain heterogeneous player parameters. Table 3.13 shows the heterogeneous player parameters which define these intervals and compares the parameter values for default players to the range of parameter values for heterogeneous players. The maximum speed of a default player, for example, is equal to the value of the server parameter `player_speed_max`, whereas for a heterogeneous player the maximum speed is some value between `player_speed_max` + `player_speed_max_delta_min` and `player_speed_max` + `player_speed_max_delta_max`. In the same way, the maximum stamina increase per cycle for a heterogeneous player is the default `stamina_inc_max` plus a value between `player_speed_max_delta_min` · `stamina_inc_max_delta_factor` and `player_speed_max_delta_max` · `stamina_inc_max_delta_factor`. This thus shows that for heterogeneous players there is a trade-off between the maximum speed of the player and the stamina increase per cycle. The calculation of value ranges for the remaining player parameters is analogous and can be done using the values in Table 3.13.

The online coach for each team (see Section 3.7) is responsible for selecting the player types to use and for changing them when necessary. As long as the play mode equals `before_kick_off`[12] the coach can change player types as often as he wants. During the game however, the coach can only make `subs_max` substitutions. Furthermore, a team is never allowed to have more than `subs_max` players of the same player type on the field *simultaneously* and must always use a default player as the goalkeeper. Each time when

[10]The values for the default player parameters are equal to those shown in earlier sections.
[11]These messages contain values for server parameters, player parameters, etc.
[12]This is the case before the start of the first half and during half-time.

Player parameters		Parameters for heterogeneous players		
Name	**Value**	**Name**	**Value**	**Range**
player_speed_max	1.0	player_speed_max_delta_min player_speed_max_delta_max	0.0 0.2	1.0−1.2
stamina_inc_max	45.0	stamina_inc_max_delta_factor player_speed_max_delta_min player_speed_max_delta_max	-100 0.0 0.2	25.0−45.0
player_decay	0.4	player_decay_delta_min player_decay_delta_max	0.0 0.2	0.4−0.6
inertia_moment	5.0	inertia_moment_delta_factor player_decay_delta_min player_decay_delta_max	25.0 0.0 0.2	5.0−10.0
dash_power_rate	0.006	dash_power_rate_delta_min dash_power_rate_delta_max	0.0 0.002	0.006−0.008
player_size	0.3	player_size_delta_factor dash_power_rate_delta_min dash_power_rate_delta_max	-100.0 0.0 0.002	0.1−0.3
kickable_margin	0.7	kickable_margin_delta_min kickable_margin_delta_max	0.0 0.2	0.7−0.9
kick_rand	0.0	kick_rand_delta_factor kickable_margin_delta_min kickable_margin_delta_max	0.5 0.0 0.2	0.0−0.1
extra_stamina	0.0	extra_stamina_delta_min extra_stamina_delta_max	0.0 100.0	0.0−100.0
effort_max	1.0	effort_max_delta_factor extra_stamina_delta_min extra_stamina_delta_max	-0.002 0.0 100.0	0.8−1.0
effort_min	0.6	effort_min_delta_factor extra_stamina_delta_min extra_stamina_delta_max	-0.002 0.0 100.0	0.4−0.6

Table 3.13: Comparison between parameter values for default players and value ranges for heterogeneous players. For each player parameter on the left, the heterogeneous player parameters which define its value range are shown on the right. This makes it clear which trade-offs exist between the player parameters.

the coach substitutes a player by another player, the new player gets the initial *Stamina*, *Recovery* and *Effort* values of the corresponding player type. Table 3.14 shows the server parameters for heterogeneous player types together with their default values.

3.6 Referee Model and Play Modes

To enforce the rules of the game, the simulator includes an *automated referee* which controls a match. This referee changes the play mode of the game in different situations. Whenever the play mode changes, the automated referee announces this by sending a message to all the players. In this way each player is constantly aware of the current state of the game. Furthermore, the referee will also announce events such as a goal or a foul. Examples of situations in which the referee acts are the following:

Parameter	Value
player_types	7
subs_max	3

Table 3.14: Server parameters for heterogeneous player types together with their default values.

- A *kick-off* takes place from the center spot on the field at the start of play and after every goal. Just before this kick-off, all the players must be on their own half of the field. After a goal, the referee will therefore suspend the match for about 5 seconds to allow for this to happen. During this time the players can use the **move** command to teleport to a position, rather than running there which is much slower and consumes stamina. If a player is still on the opposite half by the time the kick-off must take place, the referee moves this player to a *random* position on his own half of the field.
- If a goal is scored the referee announces this by broadcasting a message to all the players. He then updates the score and moves the ball to the center spot. After this, he changes the play mode to `kick_off_x`, where `x` is either `l` or `r` depending on which team should perform the kick-off, and he suspends the match for 5 seconds to allow players to move back to their own half (see above).
- When the ball goes out of bounds, the referee moves it to an appropriate position and changes the play mode to either `kick_in`, `corner_kick` or `goal_kick`. In case of a 'kick in' the referee places the ball at the point on the side line where it left the field; in case of a 'corner kick' he places the ball just to the inside of the appropriate corner of the field (the exact position is determined by the value of the server parameter `ckick_margin`); for a 'goal kick' he places the ball at the front corner of the goal area at the side of the field where it passed the end line.
- During play, the referee also enforces the *offside* rule. A player is in an offside position when he is on the opponent's half of the field and closer to the opponent's end line than all or all but one[13] of the opponent players when the ball is passed to him. The crucial moment for an offside decision is when the ball is *kicked* and not when it is received: a player can be in an onside position when he receives the ball, but could have been in an offside position when the ball was kicked towards him. The referee will actually call the offside violation when the ball comes closer than `offside_active_area_size` (currently 5.0 meters) to the player that was in an offside position when the ball was played to him. After the call he will give a free kick to the opposite team.
- When a 'kick off', 'free kick', 'kick in' or 'corner kick' has to be taken, the referee removes all the players that are located within a circle with radius `offside_kick_margin` (currently 9.15 meters) centered on the ball and places them on the perimeter of that circle. In case of an offside call, all the offending players are also moved to an onside position. Furthermore, when the play mode equals `goal_kick`, the opponent players are moved outside the penalty area and are not allowed to re-enter this area while the goal kick takes place.
- When the play mode equals `kick_off`, `free_kick`, `kick_in` or `corner_kick`, the referee changes the play mode to `play_on` immediately after the ball starts moving as a result of a **kick** command. In case of a goal kick, the play mode is changed to `play_on` as soon as the ball leaves the penalty area.
- At the end of the first and second half the referee suspends the match. Each half lasts for `half_time` (currently 300) seconds (=3,000 cycles) and when the scores are tied at the end the match is extended[14]. In this case, the team that scores the first goal in extra time wins the game (this procedure is also known as 'sudden death' or 'golden goal').

[13]This 'one' is usually the opponent goalkeeper.
[14]This actually only happens during the knock-out stage of an official competition.

Each message that is sent by the referee has the format '(**referee** *String*)', where *String* is a string denoting the contents of the message (e.g. a play mode). Players receive messages from the referee as **hear** messages. The referee messages are treated as privileged in the sense that a player can hear them in every situation independent of the number of messages that he has heard from other players. All the possible referee messages (including play modes) are shown in Table 3.15. Table 3.16 shows the server parameters which are important for the referee model together with their default values.

Message	T_c	Subsequent play mode	Comment
`before_kick_off`	0	`kick_off_`*`Side`*	at beginning of a half
`play_on`			during normal play
`time_over`			after the match
`kick_off_`*`Side`*			announce start of play (after pressing the *Kick Off* button)
`kick_in_`*`Side`*		`play_on`	play mode changes after ball kicked
`free_kick_`*`Side`*		`play_on`	play mode changes after ball kicked
`corner_kick_`*`Side`*		`play_on`	play mode changes after ball kicked
`goal_kick_`*`Side`*		`play_on`	play mode changes once ball leaves the penalty area
`drop_ball`	0	`play_on`	occurs when ball is not put into play after `drop_ball_time` cycles
`offside_`*`Side`*	30	`free_kick_`*`OSide`*	free kick for opposite side
`goal_`*`Side`*`_`*`n`*	50	`kick_off_`*`OSide`*	announce *`n`*th goal for *`Side`*
`foul_`*`Side`*	0	`free_kick_`*`OSide`*	announce foul committed by *`Side`*
`goalie_catch_ball_`*`Side`*	0	`free_kick_`*`OSide`*	announce goalie catch by *`Side`*
`time_up_without_a_team`	0	`time_over`	sent if there was no opponent until end of second half
`time_up`	0	`time_over`	sent at the end of the match (if time $\geq$ 2·`half_time` **and** scores for both teams are different)
`half_time`	0	`before_kick_off`	end of first half
`time_extended`	0	`before_kick_off`	end of second half with scores tied

Table 3.15: Possible referee messages (including play modes). Here *`Side`* is either `l` or `r` denoting the left or right team; *`OSide`* is the opposite side; T_c is the time (i.e. number of cycles) until the subsequent play mode will be announced.

Parameter	**Value**	**Parameter**	**Value**
`ckick_margin`	1.0	`forbid_kick_off_offside`	true
`offside_active_area_size`	5.0	`half_time`	300
`offside_kick_margin`	9.15	`drop_ball_time`	200
`use_offside`	true		

Table 3.16: Server parameters which are important for the referee model along with their default values.

3.7 Coach Model

The coach is a privileged client that can be used by a team to assist its players. The *soccer server* provides two kinds of coaches: an *online coach* and a *trainer*. Both coaches receive noise-free global information about all the objects on the field. Furthermore, they can substitute player types and broadcast messages to the players. The main difference between the two coaches is that the online coach may connect to official games, whereas the trainer may only be used during the development stage of a team. In general, the trainer can exercise more control over the game than the online coach can. For example, the trainer can control the play mode of the game (even deactivating the automated referee if necessary) and he can move the ball and players from both teams to any location on the field at any moment and set their directions and velocities. The trainer is therefore often used to automatically create training situations. The online coach, on the other hand, is used during games to provide additional advice and information to the players. His capabilities are limited compared to those of the trainer since he cannot control the game and is only allowed to communicate with the players of his own team. In order to prevent the online coach from controlling his players in a centralized way, the communication from the coach to the players is restricted. Since the coach receives a noise-free and global view of the field and has less real-time demands, he can spend more time deliberating over strategies than the players. The online coach is therefore a good tool for analyzing the strengths and weaknesses of the opponents and for giving strategic advice.

In order to enable a coach to work together with different teams, a standard coach language has been developed. During play, the coach is only allowed to communicate with his players using this language. The standard coach language is based on several low-level concepts which can be combined to construct higher-level concepts. It contains five types of messages:

- **Info**. Info messages contain information that the coach believes the players should know, e.g. frequent positions or player types of opponents players.
- **Advice**. Advice messages tell the players what the coach believes they should do. This can be either at an individual, group or team level. Advice messages consist of a *condition* and a *directive* that specify in which type of situation a certain action should be performed.
- **Define**. Define messages introduce names to facilitate future reference (e.g. shortcuts for regions).
- **Meta**. Meta messages contain meta-level information about the interaction between the coach and the players, such as the number of messages sent or the version of the coach language.
- **Freeform**. Freeform messages can only be sent by the coach during non-`play_on` play modes. There is no restriction on the format of these messages except for the fact that their length may not exceed `say_coach_msg_size` characters. Note that *freeform* messages will probably not be universally understood if the coach has to work with different teams.

The coach is allowed to send `clang_Type_win` messages of type `Type` (where `Type` equals `info`, `advice`, `define` or `meta`) every `clang_win_size` simulation cycles. With the current server parameter values, this means that the coach can send at most one message of each type every 300 cycles. Furthermore, the messages are delayed for `clang_mess_delay` cycles before they are heard by the players. Note that these restrictions do not apply for non-`play_on` play modes during which `clang_mess_per_cycle` messages can be sent to the players in each cycle. These messages are heard by the players without delay and do not count for the message number restriction. For *freeform* messages the only restriction is that the coach is allowed to send `say_coach_cnt_max` of them throughout the game. For a full grammar of the standard coach language we refer the reader to the *soccer server* manual [32]. Table 3.17 shows the server parameters which are important for the coach model along with their default values.

Parameter	Value	Parameter	Value
coach_port	6001	clang_win_size	300
olcoach_port	6002	clang_info_win	1
say_coach_msg_size	128	clang_advice_win	1
say_coach_cnt_max	128	clang_define_win	1
player_types	7	clang_meta_win	1
subs_max	3	clang_mess_delay	50
send_vi_step	100	clang_mess_per_cycle	1

Table 3.17: Server parameters which are important for the coach model along with their default values.

3.8 Summary of Main Features

The simulator features that have been described in this chapter together provide a challenging environment in which to conduct research. The *RoboCup Soccer Server* provides a realistic domain in the sense that it contains many real-world complexities that the agents must handle. In this section we summarize the most important characteristics and challenges of the *soccer server* simulation environment.

- The *soccer server* is a pseudo real-time system that works with discrete time intervals (simulation cycles) each lasting 100ms. During this period, the agents receive various kinds of sensory observations from the server and send requests for player actions to the server. This requires real-time decision making. It is only at the end of a cycle however, that the server executes the actions and updates the state of the environment. The server thus uses a discrete action model (see Section 3.1).
- Each agent is controlled by a separate client process which enforces a distributed approach (see Section 3.1). Agents cannot communicate with each other directly, but only indirectly via the *soccer server* using **say** and **hear** protocols which restrict the communication in several ways. Furthermore, all 22 agents on the field use a single communication channel which has a low-bandwidth and is extremely unreliable (see Section 3.2.2).
- An agent has three different types of sensors: a *visual* sensor, an *aural* sensor and a *body sensor*. The visual sensor (see Section 3.2.1) provides the agent with information (distances, directions, etc.) about all the objects in his current field of view. It also works as a proximity sensor by 'seeing' objects that are close but behind the agent. The amount of information given depends on the distance to an object. All visual information is relative from the agent's perspective and can be converted into a global representation using landmarks which have been placed on and around the field (see Figure 3.2). Noise is added to the visual sensor data by quantizing the values sent by the server. The view cone of an agent has a limited width and as a result the agent only has a partial view of the world which causes large parts of the state space to remain unobserved. However, an agent can choose to trade off the frequency of visual messages against the width of his view cone and the quality of the given information. The aural sensor (see Section 3.2.2) detects spoken messages sent by other players or the coach. It has a limited range and capacity: with the current server settings each agent can hear only one message from a nearby teammate every two simulation cycles. Messages from the referee are also treated as aural messages. The body sensor (see Section 3.2.3) provides an agent with physical information such as his stamina and current speed. It also reports information about the number of actions that the agent has performed.
- An agent can perform different types of actions (see Section 3.4) which can be divided into two distinct categories: *primary* actions (**kick**, **dash**, **turn**, **catch** and **move**) and *concurrent* actions

(**say**, **turn_neck**, **change_view**, **sense_body** and **score**). During each cycle only one primary action can be executed, whereas multiple concurrent actions can be performed simultaneously with any primary action. If an agent sends more than a single primary action command during a cycle, the server randomly chooses one for execution and discards the others. There is thus no guarantee that action commands sent by an agent are ever executed and this must therefore be verified from the sensory information that is received. One of the real-world complexities contained in the *soccer server* is that noise is added to the actuator parameters in different ways.

- Sensing and acting in the *soccer server* are asynchronous (see Section 3.1): visual information arrives at 150ms intervals (with a default view cone and view quality) and agents can perform a primary action once every 100ms. Since it is crucial for an agent to perform an action whenever he has the opportunity, this means that in some cycles the agents must act without receiving new visual information. This requires their ability to predict the current world state based on past perceptions.
- The *soccer server* simulates object movement (see Section 3.3) stepwise in a simple way: the velocity of an object is added to its position, while the velocity decays by a certain rate and increases by the acceleration of the object resulting from certain action commands. To reflect unexpected movements of objects in the real world, uniformly distributed random noise is added to the movement of all objects. Besides this, the *soccer server* also models wind as a more natural form of movement noise.
- The *soccer server* prevents players from constantly running at maximum speed by assigning a limited stamina to each of them (see Section 3.4.2). When a player performs a **dash** command this consumes some of his stamina but his stamina is also slightly restored in each cycle. If a player's stamina drops below a certain threshold this will affect the efficiency of his movement.
- A new feature of *soccer server* version 7 was the introduction of heterogeneous players (see Section 3.5). Each team can choose from several different types of players with different characteristics. These player types are randomly generated when the server is started. The different player types have different abilities based on certain trade-offs with respect to the player parameters. For example, some types will be faster than others but they will also become tired more quickly.
- The *soccer server* contains an automated referee which controls the match (see Section 3.6). This referee changes the play mode of the game in different situations. Whenever the play mode changes, the automated referee announces this by sending a message to all the players. In this way, each player is constantly aware of the current state of the game. The referee messages are treated as privileged in the sense that a player can hear them in every situation independent of the number of messages that he has heard from other players.
- It is possible to define a coach agent that receives noise-free global information about all the objects on the soccer field (see Section 3.7). The coach is a good tool for analyzing the strengths and weaknesses of the opponent team and for giving advice to his players about the best possible strategy. He can also be used for automatically creating training situations during the development stage of a team and is responsible for selecting and substituting heterogeneous player types.

Chapter 4

Agent Architecture

In this chapter we present the *UvA Trilearn 2001* agent architecture. We will define the different layers that make up this architecture and explain the functionality of each layer. Furthermore, we will describe the various components of the system and the way in which these components interact. This will give the reader a clear picture of the structure of the overall system. Note that the architecture described in this chapter is the architecture for one agent and not for the team as a whole. The *UvA Trilearn 2001* agents must operate in a dynamic and real-time environment in which communication is limited and as a result there is not enough time to negotiate complex team plans. The architecture for the agents is therefore built in such a way that each agent can work independently. Team behavior then results from the combination of behaviors of individual agents. The chapter is organized as follows. In Section 4.1 we provide a general introduction to the concept of an architecture and discuss different ways in which such an architecture can be described. In Section 4.2 we then present the functional architecture of the *UvA Trilearn 2001* agents. The various components of the system are described in Section 4.3 along with their place in the overall architecture. The chapter is concluded in Section 4.4 which explains the control flow of the system and gives an example of how the different processes interact with one another over time.

4.1 Introduction

From a software engineering perspective, architecture alludes to "the overall structure of the software and the ways in which that structure provides conceptual integrity for a system" [88]. In its simplest form, an architecture describes the structure of the system's components (modules), the way in which the components interact and the structure of the data that are used by these components. As such, the architecture can serve as a framework from which abstractions or more detailed descriptions of the system can be developed. An architectural design can be represented by a number of different models [33]. Structural models represent an architecture as an organized collection of system components. Framework models increase the level of design abstraction by attempting to identify repeatable architectural design frameworks (patterns) that are encountered in similar types of applications. Dynamic models address the behavioral aspects of the system architecture, indicating how the structure or system configuration may change as a function of external events. Process models focus on the design of the process that the system must accommodate. Finally, functional models can be used to represent the functional hierarchy of a system. To represent these models, a number of different architectural description languages have been developed, some of which will be used throughout the remainder of this chapter.

From an autonomous systems point of view, an architecture can be seen as a general description of what the system looks like and how it should behave. Such a description can be given at different levels of detail. According to [116], three levels of abstraction can be distinguished in the description of an autonomous system which are each represented by their respective architecture:

- *Functional Architecture*. This architecture concerns itself with the functional behavior that the system should exhibit. It describes what the system should be capable of doing, independent of hardware constraints and environmental conditions.
- *Operational Architecture*. This architecture describes the way in which the desired behavior can be realized, taking into account the constraints on the system as well as the environment in which it has to operate. It assigns different pieces of the overall problem to different logical modules, which have the responsibility to solve their part within a certain time.
- *Implementation Architecture*. This architecture maps the operational aspects onto hardware and software modules and as such it gives a detailed description of the realization of the system.

It is important to realize that splitting the description of a system into these three architectures is quite arbitrary and by no means universal. The choice made in [116] can be seen as roughly equivalent with splitting the development of a system into an *analysis*, *design* and *implementation* part. Many researchers however divide the process into different phases and as a result distinguish different abstraction levels for architectural descriptions. Throughout this chapter, we will mainly focus on the functional aspects of the *UvA Trilearn 2001* agent architecture. Operational and implementational descriptions are omitted, since we feel that these would not contribute to the global picture that we want to provide. However, several operational and implementational aspects will be addressed in later chapters.

In the autonomous systems community there are two general ways for describing a system at the functional level: the classical *hierarchical* approach and the *behavioral* approach. The common feature in hierarchical architectures is that the system is divided into progressive levels of abstraction which are represented by different architectural layers. In this approach, the flow of information is used as the main guideline for the decomposition of the system. The lowest level in the hierarchy takes care of the interaction with the physical world through sensors and actuators. Information enters the system through the sensors and is gradually abstracted upwards to form a high-level model of the world. The knowledge present at the highest level is used to decide on the action to be performed. This action is then gradually translated into commands which are executed at the lowest level. The power of this approach is the transparent control structure of the system: it can be seen to consist of two legs where one leg represents the flow of data upwards through the different levels and the other represents the flow of commands downwards. Disadvantages are the overhead caused by maintaining the abstract world model and the rigidity of the architecture. Due to the hierarchical structure of the system, the interaction between different modules is restricted to adjacent layers. Each module has to communicate with the layers directly above and below it through well defined interfaces. Whenever the interface of a module has to be changed, these changes will thus affect other modules as well and this might force a major redesign of the overall system.

In systems with a behavioral decomposition the main idea is to break up the problem into goals that should be achieved instead of stages of information flow. Instead of one path through which the data flow, multiple parallel paths are exploited which are called behaviors. For each behavior, the data flow along the path from sensors to actuators thereby omitting the reasoning step [9]. A behavior can thus be seen as a module which accepts sensory input and generates actuator control commands. The power of this approach is that it leads to robust systems which can be easily extended. If a single module breaks down, this will lead to a minor degradation of the overall system and not to total failure. Furthermore, multiple goals can co-exist in the system since each module is independent of the others and adding new goals will thus be easy. Drawbacks of this approach are its inefficiency and interpretability: a lot of equivalent processing and computation work is performed in several modules and it is not clear in advance how the different control signals will be combined. Besides this, the implementation is also difficult.

The two views described above can be summarized as follows. In the hierarchical approach it is first decided what the best possible strategy is, after which this chosen strategy is worked out. In the behavioral approach all possible alternative strategies are worked out and subsequently it is decided which strategy is the best. In practice however, it is often the case that neither a purely hierarchical nor a purely behavioral decomposition of an autonomous system will work well. The key observation for this is that high-level reasoning is completely sequential, whereas the real-time control of the system involves mostly parallel processing. Another type of decomposition has therefore emerged which combines the characteristics of both views into a *hybrid* approach. The general trend in hybrid architectures is that the higher abstraction levels are more hierarchical, whereas the lower levels are more behavioral. The resulting system then enables both high-level reasoning and low-level reflexive control.

4.2 Functional Architecture

Complex tasks, such as simulated robotic soccer, can always be hierarchically decomposed into several simpler subtasks. This naturally leads to agent architectures consisting of multiple layers. Figure 4.1 shows the *UvA Trilearn 2001* agent architecture. The design is based on the hybrid approach that was described in Section 4.1. This choice was made because we wanted our agents to be capable of reasoning about the best possible action without losing too much time on sending or receiving data. In order to deal with the timing constraints caused by the real-time nature of the domain, it was therefore desirable that the agents could perform this high-level reasoning process independently from taking care of their interaction with the environment. Given these characteristics, it was clear that a hierarchical decomposition was needed for the higher abstraction levels, whereas the bottom level required the benefit of parallel processing. Adopting a somewhat hybrid approach thus seemed to be the most appropriate choice.

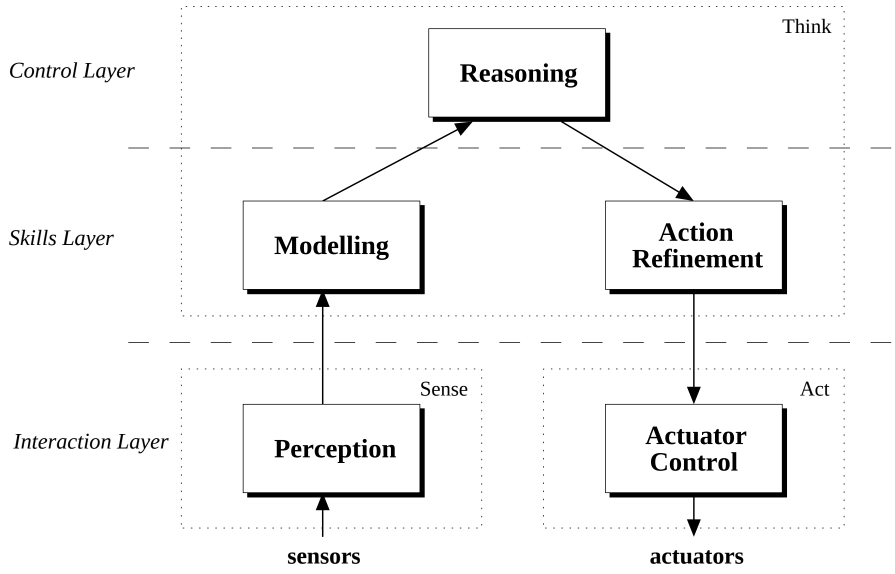


Figure 4.1: The *UvA Trilearn 2001* agent architecture.

The architecture shown in Figure 4.1 is hierarchical in the sense that it contains three layers at different levels of abstraction. The bottom layer is the *Interaction Layer* which takes care of the interaction with the *soccer server* simulation environment. This layer hides the *soccer server* details as much as possible from the other layers. The middle layer is the *Skills Layer* which uses the functionality offered by the *Interaction Layer* to build an abstract model of the world and to implement the various skills of each agent (ball interception etc.). The highest layer in the archictecture is the *Control Layer* which contains the reasoning component of the system. In this layer, the best possible action is selected from the *Skills Layer* depending on the current world state and the current strategy of the team. During a soccer game, perceptions enter the system through the *Interaction Layer* and flow upwards to the *Skills Layer* where they are used to update the agent's world model. The most recent world state information is then used by the *Control Layer* to reason about the best possible action. The action selected by the *Control Layer* is subsequently worked out in the *Skills Layer* which determines the appropriate actuator command. This command is then executed by the actuator control module in the *Interaction Layer*.

UvA Trilearn agents are thus capable of *perception*, *reasoning* and *acting*. The setup for the agent architecure shown in Figure 4.1 is such that these three activities can be performed in parallel. Since the agents have to operate in a dynamic real-time environment, the concurrent execution of these tasks is very important for performance reasons. We have chosen to implement the desired parallellism using *threads* as they appeared intuitively appropriate for the requirements and (assuming implementation in *C++*) have native support in the Linux/UNIX operating system. A thread can be thought of as "a set of properties that suggest continuousness and sequence within a machine" [12]. When a program is started, a process is created which operates in its own address space and has its own data. This process can be seen as a single thread of execution. In a single-threaded program, the initial process generated by the executable file does not create any additional threads and as a result all computations are performed sequentially. A single-threaded agent implementation would thus have meant a perception-reasoning-action loop in which the separate tasks could only be performed in a serial order. Given the nature of the simulation, it is clear that this would put a significant restriction on the agent's performance caused by the fact that he has to wait for slow I/O operations to and from the server which delay the execution of the loop.

Instead of a single thread however, a process can also have multiple threads which share the same address space and perform different operations independently. Our agent architecture is multi-threaded in the sense that it allows the agent to use a separate thread for each of his three main activities: the *Sense* thread represents the perception module, the *Act* thread represents the actuator control module and the *Think* thread represents the modules from the *Skills Layer* and *Control Layer* in Figure 4.1. The main advantage of this approach is that the delay caused by I/O to and from the server is kept to a minimum. The threads that perform I/O, i.e. the *Sense* thread and the *Act* thread, are only active when necessary and can use *blocking I/O*[1] without wasting valuable computation time for the *Think* thread. In this way the *Think* thread gets the maximum amount of CPU-time and the agent can thus spend the majority of his time thinking about the best possible action [36]. Prior research in the *soccer server* simulation environment has shown that a multi-threaded approach clearly outperforms a single-threaded one in terms of efficiency and responsiveness (see [52]). Furthermore, the division into separate threads also leads to modular programs in which the different tasks of the agent are clearly separated. There are some disadvantages as well however, since multi-threaded implementations are often more complicated than single-threaded ones due to synchronization protocols and possible deadlocks and race conditions.

[1] Both Linux and UNIX provide five different I/O models: *blocking I/O*, *non-blocking I/O*, *I/O multiplexing*, *signal-driven I/O* and *asynchronous I/O*. *Blocking I/O* means that when a process requests data, it is put to sleep until data are available.

4.3 System Components

It has already been mentioned in Section 4.1 that an architectural design can be represented by a number of different models. Figure 4.1 shows the functional hierarchy of the *UvA Trilearn 2001* agent architecture and therefore it can be seen as a functional model. In such models, the hierarchical decomposition is guided by the flow of data through the system. From a software point of view however, it is also possible to represent an architecture as an organized collection of system components. This type of model is called a structural model, since it identifies the various components of the system and specifies how these components interact with one another. This view of a system as a collection of components rather than functions is closely associated with *object orientation*, the software development paradigm that we have consequently followed throughout the project. In this paradigm objects can be seen to model almost any identifiable aspect of a problem domain: external entities, occurrences, roles, organizational units, places and structures can all be represented as objects. The object-oriented approach defines objects as program components that are linked to other components through well-defined interfaces. These objects can be categorized into classes and class hierarchies. Each class contains a set of attributes that describe the objects in the class and a set of operations that define their behavior. Objects can thus be seen to *encapsulate* both data and methods that are used to manipulate these data. The concept of *inheritance* enables the attributes and operations of a class to be inherited by all its subclasses in the class hierarchy and thus by all the objects that are instantiated from these subclasses. Consistently following an object-oriented approach will thus lead to modular systems which can be easily extended through the use of inheritance and in which the various components (i.e. objects) are reusable as a result of encapsulation. For a detailed account of the object-oriented design methodology we refer the reader to [68].

Clearly, an object-oriented design lays the foundation for a component-based model of a system architecture. In order to represent such a model one can use a number of different architectural description languages. An example of such a language is the *Unified Modeling Language* (UML) [83]. UML is a language for specifying, visualizing, constructing and documenting the artifacts of software systems. In addition, the language can also be used for modeling business processes and other types of non-software interactions. UML represents a collection of the best engineering practices that have proven successful for the modeling of large and complex systems. The language has been accepted by the Object Management Group (OMG) as a standard for modeling object-oriented programs and can be seen as the proper successor to the object modeling languages of three previously leading object-oriented methods (Booch, OMT and OOSE). UML defines nine different types of graphical diagrams[2], each of which provides a different perspective of the system under analysis or development. One of these types is the *class diagram* which describes the static structure of the system. In particular, it describes the things that exist in the system (classes, types, etc.), their internal structure and their relationships to other things. Classes are represented by rectangles which are divided into three compartments: the top compartment holds the name of the class, the middle compartment holds a list of attributes that describes the class and the bottom compartment holds the operations defined for these attributes. Relationships between classes are indicated by different types of connections (lines, arrows, etc.) which indicate the kind of relation. Relationships which are relevant for the subsequent discussion are the following:

- *Association*. This represents a static relationship between two classes. An association is depicted by a filled arrow which indicates the direction of the relationship. It is important to realize that the arrow in this case does not represent the flow of data between components, but can be seen to indicate which component needs to 'know' about the other. Multiplicity notations are placed near the ends of an association to indicate the number of instances of one class linked to one instance

[2]These are: class diagrams, object diagrams, use case diagrams, sequence diagrams, collaboration diagrams, statechart diagrams, activity diagrams, component diagrams and deployment diagrams.

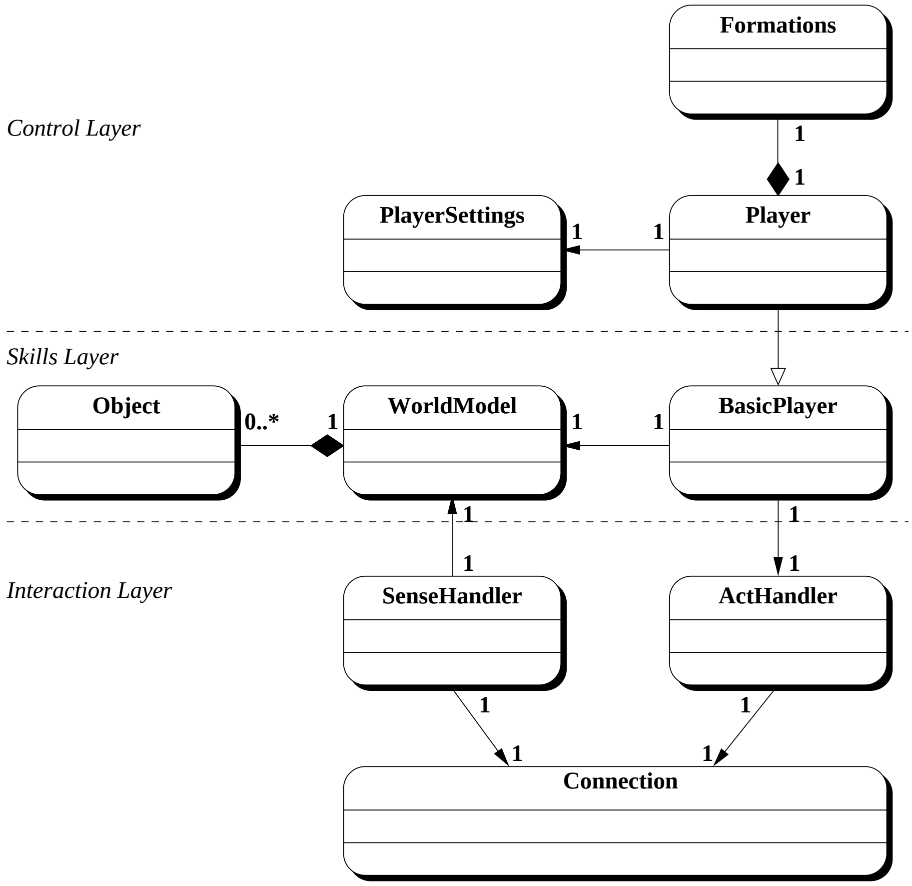


Figure 4.2: UML class diagram showing the main components of the *UvA Trilearn* agent architecture.

of the other class. For example, one company will have one or more (1..*) employees, but each employee will only work for one company.

- *Composition*. This represents a relationship between two classes in which one class is a part of the other. A composition is depicted by a solid line which ends in a filled diamond. This diamond is connected to the 'whole' class, i.e. the class that contains the other class as its part. Multiplicity relations are again placed at the ends of the connection.

- *Generalization*. This is another name for inheritance. It represents an 'is a' relationship between two classes where one class is a specialized version of the other. Generalization is depicted by an open arrow pointing from the specialized class (the subclass) to the general class (the superclass).

Figure 4.2 represents the *UvA Trilearn 2001* agent architecture in the form of an UML class diagram. The attributes and operations for each class have been omitted since this information would not contribute to the global picture that we want to present. Note that the interaction between different components has been kept to a minimum in order to keep the response time for the agents as low as possible. Furthermore, this gives the system a relatively simple structure which enables one to maintain a clear view of the different objects and the way they interact. The diagram shown in Figure 4.2 maps the various components of the system onto the functional architecture presented in Figure 4.1. The mapping is roughly as follows.

The perception and actuator control modules in the *Interaction Layer* are represented by a *SenseHandler* object and an *ActHandler* object respectively. The modeling module in the *Skills Layer* is represented by objects from the classes *WorldModel* and *Object*, whereas the action refinement module in this layer is represented by a *BasicPlayer* object. Finally, the reasoning module in the *Control Layer* is represented by a *Player* object, a *PlayerSettings* object and a *Formations* object. The role in the overall architecture of the components depicted in Figure 4.2 is explained below. Issues related to the way in which these components have been implemented are discussed in Appendix A.

- *Connection*. This component creates a UDP socket connection with the *soccer server* and contains methods for sending and receiving messages over this connection. All the communication with the *soccer server* goes through this component.
- *SenseHandler*. This component handles the processing of messages that the agent receives from the server. It parses these messages and sends the extracted information to the *WorldModel* component.
- *WorldModel*. This component contains the current representation of the world as observed by the agent. This representation includes information about all the objects on the field such as the positions and velocities of all the players and the ball. Furthermore, information concerning the current play mode is also stored, as well as the time and the score. The *WorldModel* component contains four types of methods that deal with this information in different ways:
  - *Retrieval* methods: for directly retrieving information about objects in the agent's world model.
  - *Update* methods: for updating the agent's world model based on new sensory information received from the *SenseHandler*.
  - *Prediction* methods: for predicting future states of the world based on past perceptions.
  - *High-level* methods: for deriving high-level conclusions from basic information about the state of the world (e.g. determining the fastest teammate to the ball).
- *Object*. This component contains information about all the objects in the simulation. Its implementation is spread over six separate classes, five of which have not been depicted in Figure 4.2 for reasons of space and clarity. The *Object* class is the abstract superclass that contains estimations for the global positions of all the objects and defines methods for retrieving and updating this information. This class has two subclasses: *FixedObject* and *DynamicObject*. The *FixedObject* class contains information about the stationary objects on the field (lines, flags and goals). The *DynamicObject* class contains information about moving objects and as such it adds velocity information to the general information provided by the *Object* class. The *DynamicObject* class again has two subclasses, *PlayerObject* and *BallObject*, which respectively contain information about players (teammates and opponents) and the ball. Finally, the *PlayerObject* class has one subclass called *AgentObject* which contains extra information about the agent himself. Note that the information in each class is stored together with a confidence value that indicates the reliability of the estimation. This confidence value is related to the time that passed since the object was last observed.
- *Formations*. This component contains information about possible team formations and a method for determining a strategic position. Its implementation is spread over three separate classes, two of which have not been depicted in Figure 4.2 for reasons of space and clarity. The *Formations* class contains the information about the different types of team formations and stores the current formation type and the agent's number in this formation which determines his role. Furthermore, it contains a method for determining a strategic position on the field depending on the current formation and the position of the ball. The *FormationTypeInfo* class contains all the information about one specific formation, i.e. the number of possible team formations is equal to the number of objects instantiated from this class. For each formation, the information is read from a configuration

file when the agent is started. This information includes the type of formation (4-3-3, 4-4-2, etc.), the home position for each player and the player type corresponding with each position (wing defender, central attacker, etc.). A strategic position for a player is determined by taking into account his home position in the formation and several ball attraction factors which are different for each player type. The ball attraction factors for one specific player type are defined in the *PlayerTypeInfo* class along with several positional restrictions (e.g. a central defender must stay behind the ball).

- *PlayerSettings*. This component contains the values for several player parameters which influence the agent's reasoning process and defines methods for retrieving and updating these values. Most of the parameters in the *PlayerSettings* class are threshold parameters for deciding whether a particular type of action should be performed. These parameters thus define the behavior of the agent.
- *Player*. This component contains methods for reasoning about the best possible action in a given situation. Action selection is based on the most recent information about the state of the world as obtained from the *WorldModel* component and on the role of the agent in the current team formation. For making the final decision on whether a particular type of action should be performed, the agent uses the parameter values which are specified in the *PlayerSettings* class.
- *BasicPlayer*. This component contains all the necessary information for performing the agent's individual skills such as intercepting the ball or kicking the ball to a desired position on the field. Note from Figure 4.2 that the *BasicPlayer* class is a superclass of the *Player* class. In general, the *Player* component decides which action is to be performed and this action is then subsequently worked out by the *BasicPlayer* component which uses information from the *WorldModel* component to determine an appropriate actuator command. This command is then sent to the *ActHandler* component which is responsible for the execution.
- *ActHandler*. This component is responsible for the execution of actuator commands which it receives from the *BasicPlayer* component. Each time when a command arrives, it is stored in one of two possible lists:
  - The *Primary Command List* can contain only a single command of a type which can be executed once during a cycle. Examples of such commands are **kick**, **dash**, **turn**, **catch** and **move**.
  - The *Concurrent Command List* can contain several commands of a type which can be executed multiple times during a cycle and concurrently with the commands in the *Primary Command List*. Examples of such commands are **say**, **turn_neck** and **change_view** (see Section 3.4.9).

  If the *Primary Command List* already contains a command by the time that a new command for this list is received, the old command is overwritten by the new one. A command in the *Concurrent Command List* is only overwritten if a new command of the same type is received. This ensures that the commands in these lists are always based on the most recent information about the state of the world. During a cycle, the *ActHandler* component stores each command sent by the *BasicPlayer* into the appropriate list until it receives a signal that indicates that the currently stored commands must be executed. Each command is then converted into a string message which is accepted by the *soccer server* and subsequently these messages are sent to the server using the *Connection* component.

Besides these main components we have also implemented several auxiliary classes which are used by most components shown in Figure 4.2. The functionality of each of these classes is shortly discussed below.

- *Parse*. This class contains several static methods for parsing string messages and is mainly used by the *SenseHandler* component which handles the processing of messages from the *soccer server*. The methods in this class can skip characters up to a specified point and convert parts of a string to integer or double values. The main reason for creating this class was because the standard parsing mechanism available in *C++*, the *sscanf* function, was too complex and therefore too slow. This is mainly caused by the fact that *sscanf* can only interpret a complete string and not only parts of it. The methods in our class are more specific however, and can process characters one by one without having to read the entire string. When one has to read an integer from a string, for example, all characters representing a digit are processed until a non-numeric character is encountered. We performed several experiments in which we compared our parsing methods to *sscanf* and to those of *CMUnited* [91]. The results showed that for parsing integers our method achieved a performance increase of 30.3% over the method used by *CMUnited* and 68.0% over *sscanf*; for parsing doubles the performance increase was 15.4% compared to *CMUnited* and 85.1% compared to *sscanf*. Since the agents receive many messages from the *soccer server* which all have to be parsed to extract the relevant information, these methods thus offer a considerable benefit for the processing time.

- *ServerSettings*. This class contains all the server parameters which are used for the current version of the *soccer server* ($7.x$). Examples are the maximum speed of a player (`player_speed_max`) and the stamina increase per cycle (`stamina_inc_max`). When the agent is initialized, the server sends him a message containing the values for these parameters. This message is then parsed using the methods from the *Parse* class and the resulting values are stored in the *ServerSettings* class.

- *SoccerTypes*. This class contains enumerations for different soccer types. It creates an abstraction for using soccer-related concepts (playmodes, referee messages, etc.) in a clean and consistent way throughout the code. Furthermore, this class contains methods for converting parts of string messages received from the server to the corresponding soccer types (e.g. '**(g l)**' to '`GOAL_LEFT`').

- *SoccerCommand*. This class holds all the necessary information for creating a soccer command that can be sent to the server. It contains variables denoting the possible arguments (angle, power, etc.) of the different soccer commands and stores the type of the current command. Only those variables which are related to the current type will get a legal value. Furthermore, the class contains a method for converting the command into a string message that will be accepted by the *soccer server*.

- *VecPosition*. This class contains an x- and y-coordinate denoting a position (x, y) and defines several methods which operate on this position in different ways. Methods are defined for relatively comparing positions (e.g. `isBehind`, `isBetween`, etc.) and for converting relative positions to global positions and vice versa. This class also allows you to specify positions in polar coordinates (r, ϕ) and contains a method for converting polar coordinates (r, ϕ) to Cartesian coordinates (x, y). Furthermore, the standard arithmetic operators have been overloaded for positions.

- *Line*. This class contains the representation of a line: $a \cdot x + b \cdot y + c = 0$. It allows one to specify a line in different ways: by providing three values (a, b and c), by giving two points on the line, or by specifying a single point on the line together with an angle indicating its direction. Furthermore, this class contains methods for determining the intersection point of two lines and for determining a line perpendicular to the current line that goes through a given point.

- *Rectangle*. This class contains the representation of a rectangle and contains methods that deal with rectangles. A rectangle is specified by two *VecPosition* objects denoting the upper left corner and bottom right corner respectively. The most important method in this class determines whether a given point lies inside the current rectangle.

- *Circle*. This class contains the representation of a circle and contains methods that deal with circles. A circle is specified by a *VecPosition* object which denotes its center and by a value denoting its radius. Methods have been defined for computing the area and circumference of the circle and for determining the intersection points of two circles as well as the size of their intersection area.

- *Geometry.* This class contains several static methods for performing geometrical calculations and is mainly used by the *BasicPlayer* component for working out action details. Methods have been defined for dealing with (possibly infinite) geometric series and for working with the abc-formula.
- *Timing.* This class contains a timer and methods for restarting this timer and for determining the amount of wall clock time that has elapsed since the timer was started. It is mainly used for the timing of incoming messages from the server and for debugging purposes.
- *Logger.* This class makes it possible to log various kinds of information for debugging purposes and is used by every component in the system. It allows the programmer to specify the level of abstraction from which he desires debugging information (see Appendix A.4) and contains an output stream for writing (usually a file). The *Logger* uses a *Timing* object for printing time information in the log.

4.4 Flow of Control

The control flow through the system corresponds to the natural sequence of tasks that the agent has to perform in each cycle: sense, think and act. The agent first receives sensory information which is used to create an abstract representation of the world. Based on this representation he thinks about the best possible action in the current situation and this action is subsequently worked out and executed. It has already been described in Section 4.2 that we use separate threads for each of these three tasks[3]. The *Sense* thread listens for information from the server and parses the incoming messages. The *Think* thread processes this information and comes up with an appropriate action. The *Act* thread is then responsible for sending an action command to the server. These threads can be seen to fit into the system architecture shown in Figure 4.2 in the following way: the *Sense* thread is represented by the *SenseHandler* component, the *Act* thread is represented by the *ActHandler* component and the *Think* thread is represented by the components in the *Skills Layer* and *Control Layer*.

The main advantage of this multi-threaded approach is that it minimizes the delay caused by I/O to and from the server and this allows the agent to spend the majority of his time thinking about his next action. With respect to the control flow however, the multi-threaded approach makes the implementation more complicated since the cooperating threads need to correctly synchronize with each other. The main problem is caused by the fact that the *Sense* thread and the *Think* thread need exclusive access to the data in the agent's world model. The *Sense* thread parses the information from the server and adds it to the world model. The *Think* thread then updates the world model based on the new information and uses it to reason about the next action. However, if both threads would be able to access the data concurrently, this might cause the *Think* thread to base its reasoning on an inconsistent world state (e.g. if the *Sense* thread is still busy writing to the world model when the *Think* thread starts updating it). It is therefore important to ensure that both threads do not access the world model simultaneously. One possible way to achieve this is by means of a mechanism called a *semaphore* [2]. A semaphore can be seen as a type of variable for which two operations have been defined: *SIGNAL* and *WAIT*. A semaphore is typically represented by an integer and a queue. The definitions of the semaphore operations are as follows:

- *WAIT(semaphore)*: if the value of the semaphore is greater than zero then decrement it and allow the thread to continue, else suspend the thread (i.e. it blocks on this semaphore).
- *SIGNAL(semaphore)*: if there are no threads waiting on the semaphore then increment it; else free one of the waiting threads which continues at the instruction after its *WAIT* instruction.

[3]This idea is based on work described in [52]. The main difference is that we use a more sophisticated agent-environment synchronization method to determine when to send an action to the server. This is explained in more detail in Chapter 5.

In order to achieve exclusive access to a shared resource, it is possible to use a binary semaphore (mutex) whose value is initialized to one. A *WAIT(mutex)* operation is then executed when a thread wants to start using the resource and a *SIGNAL(mutex)* operation is executed once it has finished. Consider the following example. A thread wants to access a currently unused resource that is protected by a mutex variable and thus executes a *WAIT(mutex)* operation. As a result, the value of the mutex is decremented to zero and the thread can proceed. A second thread now wants to access the same resource and thus also executes a *WAIT(mutex)* operation. However, since the value of the mutex now equals zero this thread blocks and is put into the queue. The same will happen for any other thread that wants to access the resource while the first thread is still using it. As soon as the active thread finishes using the resource, it executes a *SIGNAL(mutex)* operation which causes one of the waiting threads to be freed from the queue[4]. By the time that the queue is empty, the *SIGNAL(mutex)* operation which is executed by the last active thread will cause the value of the mutex to be restored to one until a new thread wants to access the resource, etc. Note that it is important to make sure that the active thread cannot block while it has access to the resource, since this will prevent the other threads from using it and might lead to deadlocks.

Algorithm 4.1 shows a pseudo-code implementation for the three threads used by the agent program. The world model data are protected by a binary semaphore called *lock* which has been initialized to one. The procedure is as follows. When the agent program is started, a process is created which can be seen as a single thread of execution: the *main* thread. In our implementation this *main* thread is what we call the *Think* thread. The *Think* thread starts by creating two other threads, the *Sense* thread and the *Act* thread, which are responsible for performing I/O operations. After this, the *Think* thread blocks until it receives a *READY* signal from the *Sense* thread to indicate that the world model contains new information. Meanwhile, the *Sense* thread waits for new information from the server. Once it receives this information it determines a time t after which the *Act* thread must send one or more action commands to the server and it sets a *SEND* signal to go off after this time (synchronization with the server is discussed in Chapter 5). The *Sense* thread then executes a *WAIT* operation to obtain the *lock* and once it has access it parses the information from the server and adds it to the world model. After this, it releases the lock by executing a *SIGNAL* operation and it sends a *READY* signal to the *Think* thread to indicate that new information is available. Note that the *Think* thread will always wait for this signal before determining an action, since it is useless to repeatedly do so while the contents of the world model remain the same.

When the *READY* signal arrives the *Think* thread executes *WAIT*, updates the agent's world model and determines the next action. Subsequently, it sends the appropriate actuator commands to the *Act* thread and releases the lock by executing a *SIGNAL* operation. By the time the *SEND* signal goes off, the *Act* thread converts the action commands in the *Primary* and *Concurrent Command Lists* (see Section 4.3) to string messages which are accepted by the server and it sends these messages. Note that no semaphores are used to protect the command lists since the *Act* thread must always be granted immediate access to read these lists once it has received a *SEND* signal. Also note that a new action is determined each time when sensory information arrives from the server. It is thus not the case that the agent determines a plan consisting of a sequence of actions over a number of cycles[5]. Instead, the agent determines a new actuator command in each cycle depending on the current world state. This form of coordination between reasoning and execution is referred to as *weak binding* [116] and ensures that the *Primary Command List* will always contain a command that is completely based on the most recent information about the world.

The control flow through the system can be visualized by an UML *sequence diagram*. Such a diagram describes the interaction between different objects in terms of an exchange of messages over time. A

[4] Several policies exist for scheduling the wait queue: first-come-first-serve, highest-priority-first, etc.

[5] Note that this would seriously complicate the way in which the command lists in the *ActHandler* component should be organized. One would need a way to determine whether the previously chosen plan is still useful at the current time and if not in which way the command lists should be adapted. Nevertheless, several soccer simulation teams from recent years have been reported to use such methods [13, 90].

```
{Think thread}
Create Sense thread
Create Act thread
while server_is_alive do
  block until READY signal arrives or 3 seconds have passed
  if 3 seconds have passed then
    server_is_alive = false
  else
    WAIT(lock)
    update world model
    determine next action
    send commands to Act thread
    SIGNAL(lock)
  end if
end while

{Sense thread}
while true do
  block until server message arrives
  determine send time t for Act thread        // synchronization: see Chapter 5
  set SEND signal to go off at time t
  WAIT(lock)
  parse server message and send it to world model
  SIGNAL(lock)
  send READY signal to Think thread
end while

{Act thread}
while true do
  block until SEND signal arrives
  convert commands to string messages
  send messages to server
end while
```

Algorithm 4.1: Pseudo-code implementation for the *Think*, *Sense* and *Act* threads.

sequence diagram has two dimensions: the vertical dimension represents time and the horizontal dimension represents different objects. The presence of an object over time is shown as a vertical dashed line which is called a *lifeline*. The time that an object needs to perform a certain task (either directly or through a subordinate procedure) is referred to as an *activation*. An activation is shown as a tall rectangle whose top is aligned with the starting time of the corresponding task and whose bottom is aligned with its completion time. Communication between two objects is called a *stimulus* and is depicted by a horizontal arrow pointing from the lifeline of one object to the lifeline of another object. Different kinds of communication are specified by different types of arrows. A solid arrow with a filled arrowhead denotes a procedure call. The return of such a procedure call is indicated by a dashed arrow with again a filled arrowhead. Asynchronous communication[6] between two objects is represented by a solid line with a half arrowhead.

Figure 4.3 shows an UML sequence diagram for one simulation cycle in which the agent receives both physical and visual information from the server. Note that in both cases the same sequence of actions is

[6] An object resumes its tasks immediately after sending an asynchronous message without waiting for a response.

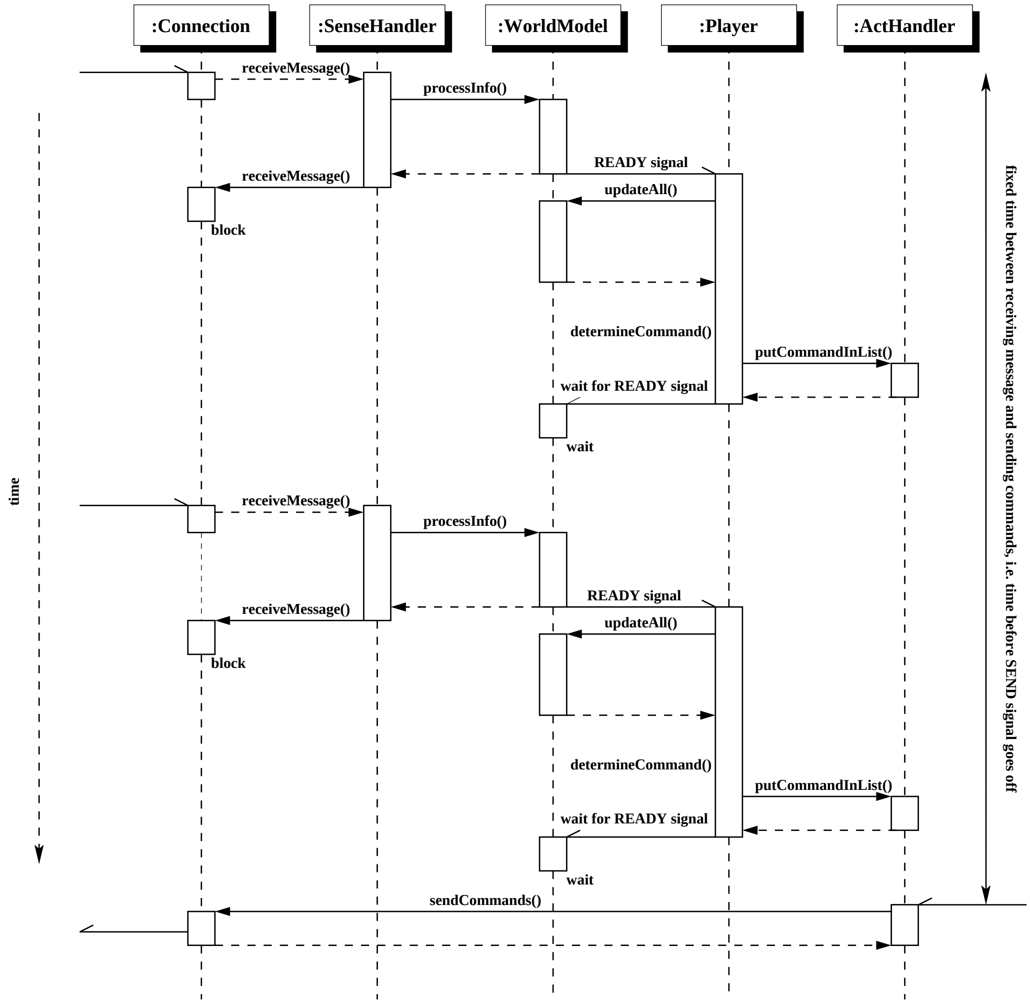


Figure 4.3: UML sequence diagram showing the interaction between different objects during a single cycle in which the agent receives both physical and visual information from the server.

performed. Activation starts when the *SenseHandler* receives a new message from the server containing physical information. This information is added to the agent's *WorldModel* after which the *SenseHandler* signals the *Player* that new information is available. The *Player* then updates the *WorldModel* and determines the next command which is sent to the *ActHandler*. After this, the *Player* blocks again until it is notified for a second time by the *SenseHandler* that new (visual) information has arrived. This happens midway through the diagram and causes the same sequence of actions to be performed. Towards the end of the cycle the *ActHandler* is then signaled to send the current command to the server. In this example, the command will be based on the information contained in the second message. It is also possible however, that the *ActHandler* is signaled while the second message is still being processed. In this case, the command sent to the server will be based on physical information (first message) which is always received at the start of a cycle. By the time that the *ActHandler* is signaled, a command will thus always be available even if the agent does not receive visual information during a cycle. In this way the agent never misses an opportunity to act.

Chapter 5

Synchronization

Synchronization between an agent and the environment he resides in is an important issue for the development of any agent. A good synchronization method can greatly enhance the agent's performance over time and in case of a team of agents it has a significant influence on the performance of the team. In this chapter we discuss the synchronization problem for the *soccer server* simulation environment and we present a comparative analysis of several agent-environment synchronization methods. The best of these methods has been used by the agents of the *UvA Trilearn 2001* soccer simulation team. It contains a flexible synchronization scheme which provides an optimal synchronization between our agents and the simulation environment. Furthermore, this method guarantees that the action chosen by an agent is always based on the latest sensory information from the server when possible. The chapter is organized as follows. In Section 5.1 we present an introduction to the problem of synchronization with the *soccer server*. Various aspects concerning the timing of incoming messages from the server are discussed in Section 5.2. In Section 5.3 we then introduce several agent-environment synchronization methods. An experimental setup for comparing these methods is described in Section 5.4 and the results of the comparative experiments are presented in Section 5.5. Finally, Section 5.6 contains a number of concluding remarks.

5.1 Introduction to the Problem of Synchronization

The *RoboCup Soccer Server* is a client-server application in which each client communicates with the server via a UDP socket. The server is responsible for executing requests from each client (i.e. agent) and for updating the environment accordingly. At specific intervals it also sends sensory information (visual, auditory and physical) about the state of the world to each agent. The agents can then use this information to decide which action they want to perform next. The *soccer server* provides a pseudo real-time simulation which works with discrete time intervals known as 'server cycles'. In the current version of the server ($7.x$) each cycle lasts for 100ms. During this period, clients can send requests for player actions which are collected by the server. It is only at the end of a cycle however, that the server executes the actions and updates the environment. The server thus uses a discrete action model. Note that each agent can only execute one primary action command (see Section 3.4.9) in each simulation cycle. When he sends multiple primary commands during a cycle, the server *randomly* chooses one for execution and discards the others. This situation will be referred to as a *clash* and must clearly be avoided. On the other hand, sending no request during a cycle will mean that the agent misses an opportunity to act and remains idle. This situation will be referred to as a *hole* and is also undesirable since in real-time adversarial domains this usually leads to the opponents gaining an advantage.

An important issue for the development of a player client is the synchronization with the *soccer server*. Since actions that need to be executed in a given cycle must arrive at the server during the right interval, a good synchronization method for sending actions to the server has a major impact on the agent's performance. The synchronization problem is not a trivial one however, due to the fact that the arrival time of messages (either from or to the server) is influenced by several factors such as the available resources (CPU-time, memory, etc.) and the speed and reliability of the network. Furthermore, determining the right moment to send an action is complicated by the fact that the agent only has explicit information about the duration of a cycle and not about its starting time. An additional problem related to synchronization is that sensing and acting in the *soccer server* are *asynchronous*. In the current server version ($7.x$) agents can send primary action commands to the server once every 100ms, but they only receive visual information at 150ms intervals[1]. Furthermore, physical information arrives every 100ms and auditory information is received at random. The challenge for the agents is to try to base their choice of action in a given cycle on the latest sensory information about the environment, while still managing to send the command to the server before the end of the cycle. In the ideal case, the agent can choose an action based on visual information about the current state of the world. This is not always possible however, since visual information is not received in every cycle and sometimes arrives too late to be able to determine an action before the cycle finishes. Furthermore, there is no specified relationship between the start of a cycle and the arrival times of sensory messages. Finding an optimal balance between the need to obtain information about the world and the need to act as often as possible is thus not a straightforward task.

In summary, the synchronization problem boils down to determining the optimal moment in each cycle to send an action to the server. In general, this is the latest possible moment which ensures that the action request will reach the server in time and which maximizes the chance of basing the chosen action on visual information about the current world state. A bad synchronization method will lead to missing action opportunities or to action choices based on old information and this will harm the agent's performance significantly. Due to the uncertain nature of many aspects of the problem, we will follow an empirical approach throughout this chapter by experimentally comparing different agent-environment synchronization alternatives. The proposed solutions are partly based on ideas introduced in [13].

5.2 Timing of Incoming Messages

At specific intervals the *soccer server* sends each agent various types of sensory information about the state of the environment. For the synchronization problem two types of server messages are important:

- **sense_body** messages arrive every 100ms and provide the agent with physical information about himself (stamina, etc.). In the remainder of this chapter these will be referred to as sense messages.
- **see** messages arrive every 150ms and provide the agent with visual information about the world.

In order to gain a better understanding of the sychronization problem it is very important to recognize the exact relationship between these two types of messages. This knowledge serves as a basis for the definition of the synchronization methods that will be presented in Section 5.3. Since this relationship depends on many uncertain factors (see Section 5.1) it cannot be derived directly from the server implementation and one has to resort to empirical methods. To this end, we created a simple test program called *message_times* which shows the arrival times of the various messages and the time differences between them. This enables one to see how the intervals between consecutive messages correspond to the interval settings (100ms and

[1] Here we ignore the possibility of trading off the frequency of visual messages against the quality of the given information and the width of the player's view cone. See Section 3.2.1.

150ms) specified in the server manual (see [32]). Two sample outputs of this program are shown below. Note that the second output was generated after the server was restarted.

```
...
sense (100): 6.61 (0.10), previous msg: 6.51
see   (100): 6.61 (0.15), previous msg: 6.46
sense (101): 6.72 (0.11), previous msg: 6.61
see   (101): 6.76 (0.15), previous msg: 6.61
sense (102): 6.82 (0.10), previous msg: 6.72
sense (103): 6.92 (0.10), previous msg: 6.82
see   (103): 6.92 (0.16), previous msg: 6.76
...
```

```
...
sense (35): 2.03 (0.11), previous msg: 1.92
see   (35): 2.07 (0.16), previous msg: 1.91
sense (36): 2.13 (0.10), previous msg: 2.03
see   (36): 2.22 (0.15), previous msg: 2.07
sense (37): 2.23 (0.10), previous msg: 2.13
sense (38): 2.33 (0.10), previous msg: 2.23
see   (38): 2.37 (0.15), previous msg: 2.22
...
```

Each of the above lines has the following format:

$$\textit{type}\ (i):\ t_{type,n}\ (t_{type,n} - t_{type,n-1}),\ \text{previous msg: } t_{type,n-1}$$

where

- *type* is the type of the message, i.e. **see** or **sense**
- i is the simulation cycle in which the message arrived (this number is contained in the actual message)
- $t_{type,n}$ is the arrival time in seconds of the nth message of type *type* relative to the starting time of the player
- $t_{type,n} - t_{type,n-1}$ is the time difference in seconds between the arrival of the nth message of type *type* and the arrival of the previous message of the same type
- $t_{type,n-1}$ is the arrival time in seconds of the previous message of the same type

It can be concluded from the output that the actual time differences between successive messages of the same type correspond closely to the interval settings in the server specification (0.1 seconds for sense messages and 0.15 seconds for see messages). Nevertheless, several small differences are visible. Every output that has been generated by the *message_times* program has shown that the intervals between consecutive messages of the same type are sometimes longer than the specified length but never shorter. It is most likely that this is caused by network traffic. In each case however, we found that the relationship between the cycle number i and the message count n for a certain message type was as would be expected from the server specification: for sense messages it held that $n = i$ and for see messages n equaled $\lceil \frac{2}{3} \cdot i \rceil$ or $\lfloor \frac{2}{3} \cdot i \rfloor$ depending on the arrival time of the first see message. The fact that the ratio between n and i remains correct despite the occasionally higher values of $t_{type,n} - t_{type,n-1}$ indicates a possible *elasticity* of the server cycles, i.e. the duration of a cycle may not always be 100ms but can exceed this value on some occasions. We will come back to this later.

A second important conclusion that can be drawn from the output generated by the *message_times* program is that during a cycle sense messages always arrive before see messages. A possible explanation for this is that the arrival of a sense message somehow indicates the start of a new cycle. We decided to further investigate this matter since knowing the starting time of a simulation cycle was obviously very important for finding an optimal solution to the synchronization problem. Since the interval between consecutive sense messages is equal to the duration of a cycle, it is clear that there must be a fixed relationship between the start of a cycle and the arrival of a sense message. In order to gain a better insight into this relationship we created a test program called *send_time_for_command* for which a pseudo-code is shown in Algorithm 5.1. The program starts up a player and waits for a sense message from the server which arrives during a cycle number divisible by ten (this can be determined by looking at the time index in the message). As soon as such a message arrives, the program waits for a specific number of milliseconds before sending a **turn** command to the server. The waiting time is then incremented by 1ms and the loop repeats itself. For reasons that will become clear later the argument supplied to the **turn** command equals the number of milliseconds between the arrival of the sense message and the send time of the command. This waiting time has been implemented in the form of a signal that is set to go off after a specific number of milliseconds. However, the current implementation of signals (and other blocking functions) in UNIX and Linux is based on the normal kernel timer mechanism which has a resolution of 10ms [2, 105]. Waiting times represented by signals will thus be rounded upwards to the first multiple of 10ms. This means, for example, that there is no difference between a waiting time of 81ms or 90ms. Nevertheless, we have chosen to increment the waiting time by 1ms after each iteration of the loop in order to make the program more general for different timer resolutions and to obtain a larger amount of useful test results.

It is possible to run the *soccer server* using an option which causes it to record all the commands sent by each client together with the numbers of the cycles in which these commands were received by the server. After running the *send_time_for_command* program this server log can be used to check for which waiting times the **turn** commands still arrived at the server during the same cycle. This will give an indication of the relationship between the arrival of a sense message and the start of a cycle. Part of the server log after running the *send_time_for_command* program on a quiet stand-alone machine[2] is shown below.

```
....
160 Recv Team_L_1: (turn 96)
170 Recv Team_L_1: (turn 97)
180 Recv Team_L_1: (turn 98)
190 Recv Team_L_1: (turn 99)
200 Recv Team_L_1: (turn 100)
211 Recv Team_L_1: (turn 101)
221 Recv Team_L_1: (turn 102)
231 Recv Team_L_1: (turn 103)
241 Recv Team_L_1: (turn 104)
...
```

The server log format is simple. Each line shows the cycle time at which the server received a command followed by the player that sent the command (in this case player number one from the left team) followed by the command that he sent. Since the argument supplied to the **turn** equals the waiting time, the output clearly shows for which values the command still arrived at the server in the same cycle. All commands were sent at cycle times divisible by ten and for waiting times up to 100ms the server received them before the end of the cycle. When the waiting time exceeded 100ms however, the commands arrived in the next cycle. Note that these results can vary greatly depending on the state of the system at the time of the experiment. We have therefore repeated the experiment 1,000 times for different system configurations. For each configuration we took three factors into account: (1) the client program was either running

[2]AMD Athlon 700MHz with 512MB running Debian Linux 2.2.17

```
for wait_time = 81 to 120 do
  sent_message = false
  while sent_message == false do
    wait for server message
    if type(message) == sense and time_index(message) mod 10 == 0 then
      wait for wait_time milliseconds
      send a (turn wait_time) command to server
      sent_message = true
    end if
  end while
end for
```

Algorithm 5.1: Pseudo-code implementation for the *send_time_for_command* program.

locally or remotely, (2) the server machine was quiet or busy (i.e. other large processes were running on it) and (3) the client machine was quiet or busy. The following configurations were tested:

- *Configuration 1.* The experiment is performed using only one machine: both the server and the client program are running on the same machine with no other large processes running. This configuration is used to test the optimal situation for client-server interaction.
- *Configuration 2.* The experiment is performed using two machines on the same network: one for the server and one for the client program. No other large processes are running on either machine. This configuration is used to test the difference in performance when a network connection is introduced.
- *Configuration 3.* The experiment is performed using two machines on the same network: one for the server and one for the client program. During the experiment, a complete team consisting of 11 player clients is running on the server machine while no other large processes are running on the client machine. This configuration is used to test the difference in performance when other processes are running on the server machine.
- *Configuration 4.* The experiment is performed using three machines on the same network: one for the server, one for 11 players belonging to the left team and one for 11 players belonging to the right team. This configuration is used to test the most common situation where the server and each team run on different machines and where the server has to send messages to all 22 players and vice versa.

Table 5.1 shows the results of the experiments for the different configurations. Note that due to the system timer resolution of 10ms the results for equivalent send times have been joined together. This means that each percentage is based on 10,000 samples. The results show that for the first configuration the commands almost always arrive at the server during the same cycle for send times up to 100ms after the arrival of a sense message. The key observation at this point is that this means that there must be at least 100ms between the arrival of a sense message and the beginning of the next cycle. Since the specified duration of a simulation cycle is also 100ms, it can thus be concluded that *the arrival of a sense message coincides with the start of a cycle.* When the client program runs on a remote machine (second configuration) the results show that the percentage of message arrivals in the same cycle is slightly lower as a result of network traffic. Even in this case however, most of the commands sent up to 100ms after the arrival of a sense message still reach the server during the same cycle. The results for the third configuration (busy server) are surprising: even commands sent up to 120ms after the arrival of a sense message still reach the server before the end of the cycle in about 30% of the cases. This implies that in some cases the duration of a cycle exceeds the specified value of 100ms which indicates a possible elasticity of the cycle length as mentioned earlier. This effect can be explained as follows. The additional client

Configuration				Time (ms)			
nr	**client**	**server**	**client**	**81-90**	**91-100**	**101-110**	**111-120**
1	local	quiet	quiet	100.00%	99.30%	0.03%	0.01%
2	remote	quiet	quiet	100.00%	96.80%	0.00%	0.00%
3	remote	busy	quiet	100.00%	96.78%	74.54%	31.32%
4	remote	quiet	busy	99.70%	79.94%	0.00%	0.00%

Table 5.1: Percentage of message arrivals in the same cycle for different system configurations and different send times. Other configurations were not relevant for the problem at hand and were therefore not tested.

processes running on the server machine all consume a certain amount of CPU-time. As a result, there is not enough CPU-time left for the server to handle all the action requests and update the environment within the specified time. Since the server will only initiate a new cycle once it has finished processing the requests from the previous one, this means that a cycle can sometimes last longer than the specified 100ms time window. Note that it is not very likely that this will happen during official competitions[3], since the fourth configuration then usually applies: the server runs on a separate machine with both teams running on remote machines. The results show that in this case almost all the commands sent up to 90ms after the arrival of a sense message reach the server before the end of the cycle. However, when the waiting time equals 100ms, 20% of the commands do not reach the server in time. Compared with the results for the second configuration, this performance decrease can be attributed to the extra CPU-load and increased network traffic caused by the additional player clients running on the client machine.

The output of the *message times* program shown at the beginning of this section demonstrates that there is an almost fixed relationship between the arrival times of see messages and sense messages. Furthermore, it is visible that this relationship is different after a restart of the *soccer server*. The first part of the output shows that see messages arrive either at the same time as sense messages or halfway between two sense messages. In the second output however, see messages arrive either 40ms or 90ms after a sense message. In general, since the intervals between consecutive sense and see messages are 100ms and 150ms respectively, the arrival times of both types of messages will describe a pattern that is repeated every 300ms (least common multiple of 100 and 150) and thus every three cycles. Clearly, see messages will only arrive in two out of these three cycles: once in the first half of the cycle and once in the second half of the cycle. Besides this, the only thing that we can be certain of is that the first see message will arrive within 150ms after the first sense message. Assume that δ denotes the time difference in milliseconds between the arrival of a sense message and the arrival of a see message in the first half of the same cycle, i.e.

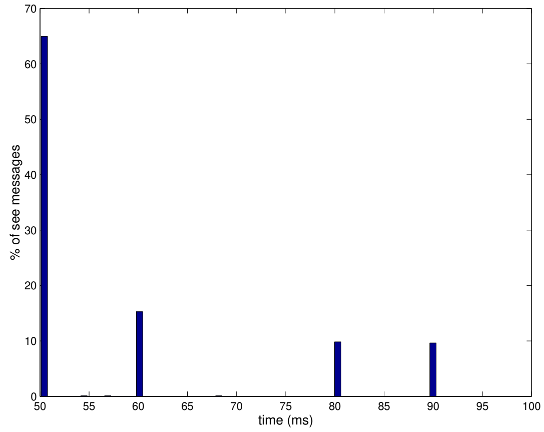
$$\delta \quad = \quad (t_{see,1} - t_{sense,1}) \mod 50 \tag{5.1}$$

The arrival times of see messages relative to the start of a cycle (and thus to the arrival times of sense messages) then repeat themselves as follows:

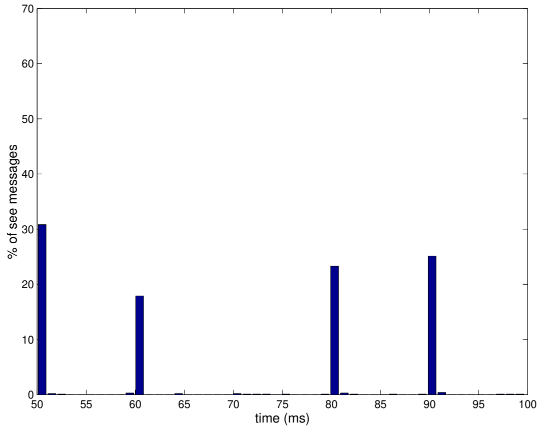
$$pattern \quad = \quad \{\delta, \delta + 50, -\} \tag{5.2}$$

where '$-$' denotes that no visual information arrives during a cycle. Note that the value for δ cannot be known in advance and will be different each time when the server is started. In order to gain a deeper insight into the possible values for δ and into the distribution of these values, an experiment was performed where the server and a single player were restarted 1,000 times and in which on each occassion the time was recorded of the first see message that arrived during the second half of a cycle (i.e. $\delta + 50$). This was done for each of the four configurations defined earlier. After this, we plotted the various $(\delta + 50)$ values

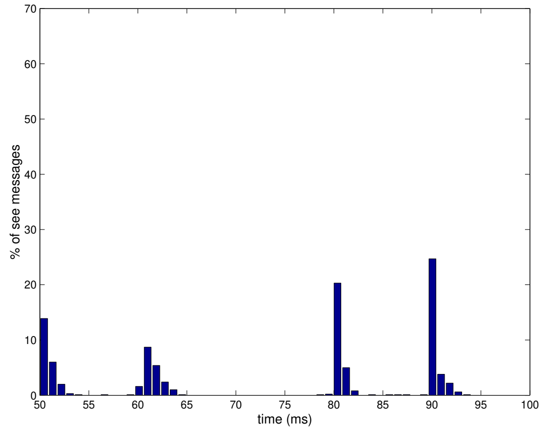
[3]It can be of influence however, when performing tests in an environment where not enough machines are available.



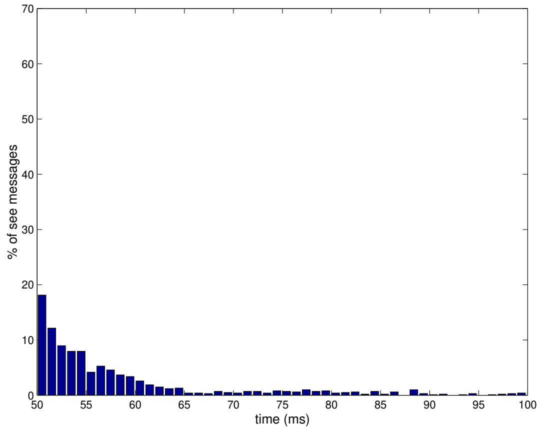
(a) local client, quiet server, quiet client



(b) remote client, quiet server, quiet client



(c) remote client, busy server, quiet client



(d) remote client, quiet server, busy client

Figure 5.1: Histograms showing the distributions of see message arrivals in the second half of a cycle.

for each server restart and transformed these plots into corresponding histograms in order to get a better view of the distribution of arrival times for each configuration. These histograms are shown in Figure 5.1.

Figure 5.1(a) shows that the distribution of see message arrivals in an ideal situation is all but uniform. The arrival times tend to concentrate around multiples of 10ms which will be referred to as 'time lines'. Note that this is probably caused by the limited timer resolution of the system[4]. It seems that the majority of see messages arrives 50ms after a sense message, i.e. midway through the cycle. Furthermore, it is visible that the server implementation is apparently such that a see message never arrives after exactly 70ms. Figure 5.1(b) shows the situation where the agent program runs on a remote machine. In this case we can see that due to the network traffic introduced by this configuration a small number of see messages does not arrive exactly at a multiple of 10ms. Furthermore, the arrival times are now

[4]Note that the timer resolution plays no role for our own measurements of $(\delta + 50)$ since we only need to compare the arrival times of see and sense messages. For this we do not need signals (i.e. no timers) since we can simply use the wall clock time.

more evenly distributed over the time lines, although the majority still arrives midway through the cycle. However, when a complete team consisting of 11 players is running on the server machine, the time lines become less explicit and the arrival times start to increase (see Figure 5.1(c)). This is caused by the extra CPU-load on the server machine as well as the fact that the server now has to send messages to 12 players instead of one. Finally, the results for the fourth configuration are slightly surprising as can be seen in Figure 5.1(d): the time lines have disappeared completely. Recall that in this case the experiment has been performed using three machines on the same network: one for the server and one for each team. It is likely that the effect which is visible in Figure 5.1(d) is caused by the increased network traffic resulting from this configuration. In addition, the arrival times will also be influenced by the extra CPU-load on the client machine which now receives server messages for 11 players instead of one. Note that since the server runs on a separate machine, the arrival times are again biased towards the midpoint of a cycle.

5.3 A Number of Synchronization Alternatives

In Section 5.2 we discussed several important issues concerning the timing of incoming messages. These issues are closely related to the synchronization problem and understanding them is a prerequisite for the design of a successful synchronization scheme. In this section we will describe four different synchronization methods which can be seen as potential solutions to the synchronization problem. The naming convention for these methods is based on the categorization of scheduling schemes introduced in [13]. Firstly, a timing mechanism will be called either *internal* or *external*. Internal timing can be thought of as a biological clock or regulation mechanism that is realized by the use of system timing routines. External timing on the other hand can be seen as the observation of change in the environment. In the *soccer server* domain this is expressed by the arrival of sensory information. Furthermore, *windowing* refers to the specification of a time window ('window of opportunity') that defines the amount of time that the agent can use to determine his next action. As soon as the time window expires he must send a command to the server.

5.3.1 Method 1: External Basic

A naive solution to the synchronization problem is to only perform an action when a visible change in the environment occurs. This means that the agent will only send an action command to the server after receiving a see message. Algorithm 5.2 shows a pseudo-code for this synchronization method to which we will refer as *External Basic*. An advantage of this approach is that it is linear (single-threaded): the agent waits for visual information, determines an action, sends a command, waits for visual information, etc. In this way, the agent can always base his next action on the most recent visual information about the world. Furthermore, during cycles in which no visual information is received he does not need the ability to predict the current world state using perceptions from the past. However, since visual information only arrives in two out of every three cycles (see Section 5.2), the agent will utilize only about 67% of his action opportunities. During the remaining 33% of the cycles the agent remains idle and this will slow him down significantly (e.g. when he is running after the ball). This is a big disadvantage since in real-time adversarial domains it usually allows the opponents to gain the initiative. Figure 5.2 shows an example of a situation in which the *External Basic* synchronization scheme is applied. In this figure time runs horizontally from left to right. Different kinds of arrows are used to indicate different types of messages from the server (top) to the agent (bottom) and vice versa. The example shows that the server sends a sense message to the agent at the beginning of each cycle which the agent receives a fraction later. As soon as the agent receives a see message he determines his next action and sends an action command back to the server. Note that no actions are sent in cycles 3 and 6 since the agent has not received any visual information in these cycles. Also note that in the given example the commands all arrive at the server before the end of the cycle (i.e. in time to be executed in the next cycle). This is caused by the fact that the agent receives see messages relatively early in a cycle which gives him enough time to determine an action and send it to the server. In general however, it is possible that see messages arrive later and as a result the action commands might not reach the server before the next cycle starts.

```
while server_is_alive do
  wait for see message
  determine next action
  send action command to server
end while
```

Algorithm 5.2: Pseudo-code for the *External Basic* synchronization method.

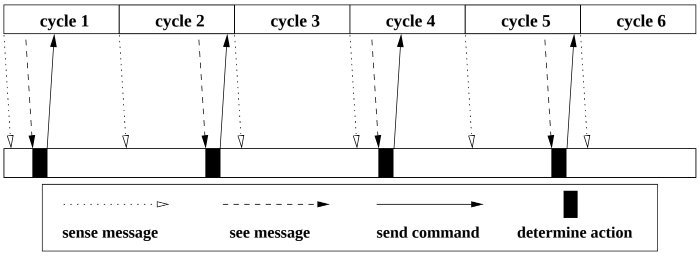


Figure 5.2: Synchronization example using the *External Basic* scheme.

5.3.2 Method 2: Internal Basic

Another possible solution to the synchronization problem is to send an action to the server every 100ms using some kind of internal clock to count these intervals. This approach will be referred to as *Internal Basic* and is shown in Algorithm 5.3. It has been implemented by installing a mechanism that sends a signal at exact 100ms intervals. Each time when such a signal arrives it is processed by a signal handler which determines an action and sends it to the server. Since the length of the interval equals the duration of a cycle, this method has the advantage that the server will receive a command in each cycle. The agent will thus utilize every action opportunity. However, due to the internal nature of the timing mechanism it cannot be known in which part of a cycle the signal arrives. In the most ideal situation, the signal will arrive towards the end of a cycle since this increases the probability that the agent can determine an action based on visual information from that same cycle. In the worst case however, the signal arrives at the beginning of a cycle and the agent will then always choose an action based on old visual information from the cycle before. Figure 5.3 shows an example of a situation in which the *External Basic* synchronization scheme is applied. The figure shows that consecutive action commands arrive at the server in adjacent cycles, i.e. no action opportunities are missed. Actions are chosen at 100ms intervals which are in no

```
{Main thread}
install signal that comes every 100ms

{Signal handler – called when signal arrives}
determine next action
send action command to server
```

Algorithm 5.3: Pseudo-code for the *Internal Basic* synchronization method.

way related to the arrival of sensory information from the server. Note that in the given example the action commands are only based on current visual information if the see message arrives in the first half of a cycle. This means that in two out of every three cycles the command will be based on old visual information about the state of the world.

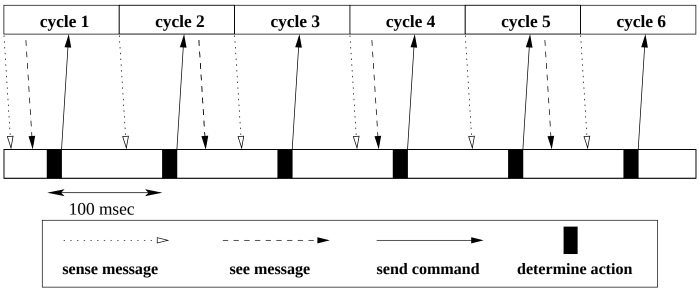


Figure 5.3: Synchronization example using the *Internal Basic* scheme.

5.3.3 Method 3: Fixed External Windowing

We have seen in Section 5.3.2 that sending an action to the server every 100ms has the advantage that no action opportunities will be missed. Clearly, the method described in that section can be improved by choosing a more appropriate time to send an action to the server. The problem with the *Internal Basic* scheme is that the time of sending is not related to the arrival of server messages which might lead to action choices based on old world state information. However, by using an external timing mechanism it is possible to relate the send time of the command to the start of a cycle. This can be done in such a way that there is a high probability that the agent can choose an action based on current visual information, while there is enough time for the action command to reach the server before the end of the cycle. These ideas have been incorporated into the *Fixed External Windowing* synchronization scheme. This method is based on the assumption that the arrival of a sense message indicates the start of a new cycle (see Section 5.2). Each time when a sense message arrives from the server, a signal is set to go off 90ms later[5]. During this time, the agent will first determine an action based on this sense message and on the most recent see message. However, if another see message arrives before the signal goes off, the agent can use the new information to make a better choice. As soon as the signal goes off, the currently chosen action is sent to the server. This is an example of an *anytime algorithm* (see [23]). Note that the send time of 90ms is based on the results presented in Table 5.1. These show that for the fourth configuration (which is most common) almost all commands sent after exactly 90ms still reach the server before the end of the cycle.

The main advantage of this approach is that during cycles in which a see message arrives it very often allows the agent to choose his next action based on visual information about the current world state. In addition, it is not very likely that action opportunities will be missed since a send time of 90ms gives a high probability that a command will reach the server before the end of the cycle. Experiments have shown that choosing an action on average takes about 5ms and it is thus only when a see message arrives more than 85ms after the start of a cycle that the agent will not have enough time to choose an action based on the new information. A disadvantage of this method is that the commands reach the server at the very end of a cycle leaving only a small margin to the start of the next one. This makes the approach sensitive to temporary increases in CPU-load or network traffic since these might cause the command to reach the server too late. This will have a negative impact on the synchronization in two ways. Firstly, the current cycle will be a *hole* since the agent misses an opportunity to act. Secondly, a *clash* will occur in

[5]Note that due to the limited timer resolution of the system any value between 81ms and 90ms will produce the same result.

the subsequent cycle since the server receives two action commands[6]. The first of these two commands is based on information from the wrong cycle and since the server randomly chooses one for execution there is a 50% chance that this action will actually be performed while the 'better' action will be discarded.

A pseudo-code implementation for the *Fixed External Windowing* method is shown in Algorithm 5.4. Note that the actual implementation becomes slightly more difficult as compared to the previous methods, since two threads are now needed: one to determine an action based on the latest see message and one for sending a command to the server after the 90ms time period has expired. Note that the first of these threads must make sure that a command is always available when the signal goes off. This command will either be based on visual information from the same cycle or on old information from the previous cycle. Figure 5.4 shows an example of a situation in which the *Fixed External Windowing* scheme is applied. The figure shows that each time when sensory information is received a new action is determined. As soon as the signal goes off, the most recently chosen command is sent to the server. Note that in the given example the agent always succeeds in sending an action based on current visual information about the world when this is possible (i.e. when a see message arrives in that particular cycle).

```
{Main thread}
while server_is_alive do
   block until server message arrives
   if type(message) == sense then
      set signal to go off after 90ms
   end if
   current_action = determine next action
end while

{Signal handler (Act thread) − called when signal arrives}
send current_action to server
```

Algorithm 5.4: Pseudo-code for the *Fixed External Windowing* synchronization method.

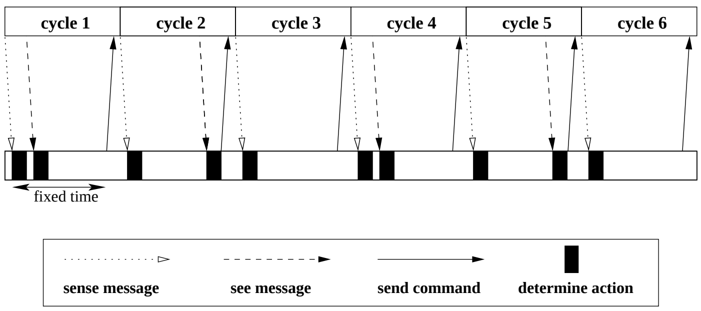


Figure 5.4: Synchronization example using the *Fixed External Windowing* scheme.

5.3.4 Method 4: Flexible External Windowing

Two important characteristics of a good synchronization method are to send an action to the server in each cycle and to base this action on the most recent world state information when possible. The method described in Section 5.3.3 exhibited both characteristics, but had the problem that the fixed send time of

[6]Assuming that the command in the subsequent cycle reaches the server in time.

90ms carried the risk of causing holes and clashes since it lay close to the end of a cycle. It is possible to improve this policy by making the send time flexible depending on the arrival times of see messages[7]. Equation 5.2 shows that the arrival times of see messages form a pattern that repeats itself every three cycles. Due to the repetitive nature of this pattern it is possible to predict in each cycle whether a see message will arrive or not, and if so, in which half of the cycle it will come. For example, if no see message arrived during a cycle then the agent knows that one will come in the first half of the next cycle and in the second half of the cycle after that. This information can be used to determine a more appropriate send time for the given situation. Clearly, there is no need to postpone sending a command until 90ms after the start of a cycle when one knows that no additional visual information will come from the server before this time. In cases where a see message will arrive during the first half of a cycle or will not arrive at all, it is sensible to use a safer value for the send time in order to make sure that the command will reach the server before the end of the cycle. A possible value is 70ms since this virtually ensures a timely arrival and gives the agent more than enough time to choose an action. The send time will only remain at 90ms in cycles during which a see message will arrive somewhere in the second half of the cycle. In this way, the agent will often be able to use this see message for determining his next action while the action command still has a good chance to reach the server in time. Note that this method can be further improved by immediately sending an available command if this command is based on visual information from the current cycle. In this case, the agent can be sure that he will not receive any more see messages during that cycle and there is thus no need to wait any longer. This will especially improve the performance when see messages arrive during the second half of a cycle, since the risky send time of 90ms can then often be avoided.

The ideas described above have been incorporated into the *Flexible External Windowing* synchronization scheme which is shown in Algorithm 5.5. It is important to realize that we have deliberately chosen not to fix the send time *exactly* to the arrival times of see messages since this could cause problems when during play the time differences between see and sense messages would change for some reason. Instead, the send time depends on the part of the cycle in which a see message will arrive. Also note that the pattern index indicating the current position in the δ-pattern (see Equation 5.2) is always reset when a see message arrives during the first half of a cycle (i.e. no see message arrived in the previous cycle). This makes the proposed solution more robust since it ensures that the execution will remain correct if for some reason two cycles would pass without receiving visual information. Figure 5.5 shows an example of a situation in which the *Flexible External Windowing* scheme is applied. The figure shows that when visual information arrives a command is determined and sent to the server immediately. Furthermore, during cycles in which no see messages are received the command is sent relatively early to avoid the risk of a hole.

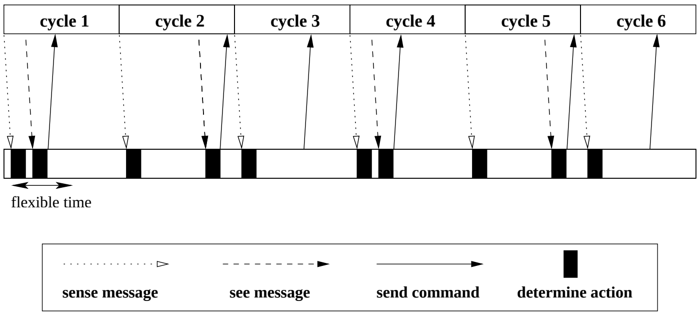


Figure 5.5: Synchronization example using the *Flexible External Windowing* scheme.

[7]This differs from the method described in [13] where a sequence of commands that have to be executed in the forthcoming cycles is stored in a queue and where the send time depends on the state of this queue (i.e. whether it is empty or not).

```
{Main thread}
pattern_index = 0                  // 0 = see in 1st half, 1 = see in 2nd half, 2 = no see
while server_is_alive do
  block until server message arrives
  if type(message) == sense then
    sent_message = false           // new cycle starts
    if pattern_index == 0 or pattern_index == 2 then
      set signal to go off after 70ms
    else if pattern_index == 1 then
      set signal to go off after 90ms
    end if
    pattern_index = (pattern_index + 1) mod 3
  else if type(message) == see then
    if no see message arrived in the previous cycle then
      pattern_index = 1            // reset pattern: see in 2nd half in next cycle
    end if
  end if
  current_action = determine next action
  if type(message) == see and sent_message == false then
    set signal to go off immediately
  end if
end while

{Signal handler (Act thread) − called when signal arrives}
if sent_message == false then
  send current_action to server
  sent_message = true
end if
```

Algorithm 5.5: Pseudo-code for the *Flexible External Windowing* synchronization method.

5.4 An Experimental Setup for Comparing the Alternatives

To compare the synchronization alternatives presented in Section 5.3, an experiment was performed which produced an empirical measurement of the success of each method. In this experiment a player was placed on the field for a period of 99 cycles during which he sent actions to the server using one of the synchronization methods discussed earlier. Along with this player, 10 additional players were determining and performing actions on the same machine whereas the server and another team consisting of 11 players were running on two separate machines. This was done to create a situation that was similar to a real match. The procedure of choosing an action was simulated by blocking the client process for a period of 10ms. This was quite high as compared to the time actually needed, but a restriction caused by the limited timer resolution of the system. The command sent in each cycle depended on the arrival time of the last see message. If it was possible to base the command on a see message that arrived during the same cycle (taking into account the 10ms needed to choose the action) the player sent a **dash** command to the server. However, if the command had to be based on visual information from the previous cycle the player sent a **turn** command. The argument supplied to both commands always equaled the cycle number of the cycle in which the command was sent. Afterwards, this made it possible to check in the server logfile whether the command had been received by the server in the same cycle. This experiment was repeated 250 times for each synchronization method. Note that in order to obtain a good indication of the success of each method, it is necessary to perform the experiment multiple times since the arrival times of see messages will be different when the server is restarted. A duration of 99 cycles for each trial was chosen for two reasons. In the first place this number is divisible by three and since the arrival times of see messages describe a pattern that is repeated every three cycles (see Equation 5.2) this means that the arrival times will be equally distributed. Furthermore, 99 cycles seemed long enough to give a good indication of a method's performance on that particular trial. After each trial the server logfile was parsed to compute several statistics for the method used. In the subsequent discussion we will use the following notation to explain the results:

- δ_i is a boolean function that returns 1 when the player sent a **dash** command in cycle i and 0 otherwise.
- τ_i is a boolean function that returns 1 when the player sent a **turn** command in cycle i and 0 otherwise.
- α_i is a function that returns the argument of the first command received by the server in cycle i; this function is undefined when no command is received in cycle i.
- γ_i is a function that returns the number of commands received by the server in cycle i.
- $\Theta(\beta)$ is a function that returns 1 when the boolean expression β is true and 0 otherwise.

Each experiment was performed in *soccer server* version 7.10 using three AMD Athlon 700MHz/512MB machines running Debian Linux 2.2.17. In the sequel we will refer to an action that was based on current visual information as an *optimized* action, whereas an action based on old visual information will be called *unoptimized*. In this experiment a **dash** command is thus an optimized action, whereas a **turn** command is unoptimized. For each method the following statistics were extracted from the server log after a trial:

- The total number of cycles n.
- The number of optimized actions received by the server in the correct cycle, i.e. $\sum_{i=1}^{n} \Theta(\delta_i \wedge \alpha_i = i)$. These actions have been based on a see message arriving in the current cycle and have reached the server before the end of this cycle. This is obviously the ideal situation.
- The number of optimized actions received by the server in the wrong cycle, i.e. $\sum_{i=1}^{n} \Theta(\delta_i \wedge \alpha_i \neq i)$. These actions have been based on a see message arriving in the current cycle, but only reached the server at the start of the next cycle.
- The number of unoptimized actions received by the server in the correct cycle, i.e. $\sum_{i=1}^{n} \Theta(\tau_i \wedge \alpha_i = i)$. These actions have been based on a see message arriving in the previous cycle (either because no see message arrived in the current cycle or because it arrived too late) and have reached the server before the end of the current cycle.
- The number of unoptimized actions received by the server in the wrong cycle, i.e. $\sum_{i=1}^{n} \Theta(\tau_i \wedge \alpha_i \neq i)$. These actions have been based on a see message arriving in the previous cycle and only reached the server at the start of the next cycle.
- The number of holes, i.e. $\sum_{i=1}^{n} \Theta(\neg\delta_i \wedge \neg\tau_i)$. This is the number of cycles in which the server did not receive an action command and thus equals the number of missed action opportunities.
- The number of clashes, i.e. $\sum_{i=1}^{n} \Theta(\gamma_i > 1)$. This is the number of cycles in which the server received more than one action command. One of these commands will be randomly chosen for execution.

	Method 1		Method 2		Method 3		Method 4	
total cycles	24750	100.00%	24750	100.00%	24750	100.00%	24750	100.00%
optimized (on time)	14899	60.20%	7333	29.63%	13039	52.68%	14415	58.24%
optimized (late)	1605	6.48%	1268	5.12%	867	3.50%	50	0.20%
unoptimized (on time)	0	0.00%	15191	61.38%	9744	39.37%	10199	41.21%
unoptimized (late)	0	0.00%	765	3.09%	326	1.32%	16	0.06%
holes	8246	33.32%	193	0.78%	774	3.13%	70	0.28%
clashes	0	0.00%	58	0.23%	742	3.00%	65	0.26%

Table 5.2: A comparative analysis of different agent-environment synchronization methods.

5.5 Results

The results of the experiment described in Section 5.4 are presented in Table 5.2. The first method was the *External Basic* scheme in which action commands are only sent after a see message. The results for this method clearly show that it is very inefficient. Since see messages will only arrive in two out of every three simulation cycles, one-third of the total number of cycles will automatically be a hole as is shown in Table 5.2. The agent will thus miss many action opportunities and this will slow him down significantly (e.g. when running after the ball). Note that all the action commands sent to the server are optimized due to the fact that an action is always chosen directly after the arrival of a see message. The percentage of commands that reach the server in the next cycle can be attributed to cycles in which the see message arrived very late. In these cases, the cycle has ended while the agent determines his next action and sends it to the server and as a result the action command arrives at the beginning of the next cycle.

The second method was the *Internal Basic* scheme in which an action is sent to the server every 100ms. Due to the internal nature of the timing mechanism, the 100ms intervals for choosing an action are in no way related to the arrival of sensory information from the server. The success of this method therefore depends heavily on the relationship between the arrival times of see messages and the send time of commands. This relationship will be different in each trial and thus so will the results. Table 5.2 shows that a large percentage of the commands is unoptimized. This can be explained as follows. Since see messages arrive only in two out of every three cycles, at least 33% of all the commands will always be unoptimized. During the remaining 67% of the cycles there is about a 50% chance that the signal that counts the 100ms intervals will go off before the see message arrives and in each of these cases the action will be unoptimized. In total, about two-thirds of the commands sent will thus be unoptimized. Furthermore, if the signal arrives late in the cycle then the command will not reach the server before the next cycle starts. Note that with this method it is not always the case that a command arriving in the next cycle leads to a hole. Since commands are sent every 100ms, it is possible that when the signal comes very late *all* the commands reach the server in the next cycle which will not cause any holes. Also note that if a hole occurs this will not always lead to a clash later on, i.e. the number of holes is not equal to the number of clashes. Due to heavy CPU-load or network traffic it is possible that successive commands reach the server slightly further than 100ms apart and this can lead to a hole. However, a clash will not necessarily follow since the subsequent send signal comes 100ms after the previous one which will cause the next command to reach the server one cycle later. Reversely, a clash can also occur without a hole due to the possible elasticity of server cycles (see Section 5.2). When for some reason a cycle lasts longer than the specified 100ms, this can cause a clash since the send times for consecutive commands will still be 100ms apart.

The third method was the *Fixed External Windowing* scheme in which the send time of a command is related to the start of a cycle (i.e. the timing mechanism is external). Using this scheme the agent always sends an action command to the server exactly 90ms after the arrival of a sense message. Compared to the second method we can see that the number of optimized commands is significantly larger now: in about 56% (out of a maximum possible 67%) of the cycles the agent performs an optimized action. The unoptimized commands come from cycles in which no see message arrived or from cycles in which the see message arrived more than 80ms after the start of the cycle (taking into account the 10ms needed for choosing an action). Furthermore, a larger number of commands now reach the server in the intended cycle due to the fact that the send time of 90ms gives a high probability that the command arrives on time (see Table 5.1). A disadvantage of this approach is that the number of holes has increased as compared to the *Internal Basic* scheme. This is because the late send time causes the commands to reach the server at the very end of a cycle and leaves only a small margin to the start of the next one. It is thus likely that temporary increases in CPU-load or network traffic will lead to holes. Note that due to the external nature of the timing mechanism the number of holes and clashes is more balanced. Since the send time of a command is now related to the start of a cycle instead of the previous send time, a hole will eventually always lead to a clash and clashes cannot occur without holes. Also note that the number of holes is not equal to the number of times in which a command reached the server in the wrong cycle. This is because a single hole can be followed by several late arrivals which will eventually be followed by a single clash.

The fourth method was the *Flexible External Windowing* scheme in which a flexible send time is used depending on the arrival times of see messages. Action commands based on current visual information are always sent immediately after they have been determined. Unoptimized actions are sent either 70ms or 90ms after the start of a cycle depending on whether visual information is not expected during that cycle or expected in the second half. The results show that compared to the third method the percentage of optimized actions is about the same. When we look at the total number of actions performed (i.e. subtract the number of holes from the total number of cycles) we can see that for both methods about 58% of these actions is optimized. However, the results for this method show that most of the commands now reach the server in the correct cycle and as a result the number of holes is significantly lower. This is caused by the fact that commands are only sent to the server at the very end of a cycle when this is necessary, i.e. when a see message arrives very late. In all other cases the command will be sent earlier which reduces the risk of a late arrival. Note that for this method the number of holes (and clashes) is almost equal to the number of late arrivals. This is because the flexible send time causes most of the commands to reach the server before the cycle ends. It is thus not likely that an occasional hole will be followed by several late arrivals before a clash occurs. Instead, if a command reaches the server at the start of the next cycle then the resulting hole will usually be directly followed by a clash. Altogether, the results in Table 5.2 clearly show that the *Flexible External Windowing* scheme performs better than the other methods.

5.6 Conclusion

In summary, a good synchronization method needs to have the following important properties:

1. It has to make sure that an action command is sent to the server in each cycle.
2. It has to make sure that the send time of each command is such that there is a high probability that the command will reach the server before the end of the cycle.
3. It has to make sure that the chosen action is based on current visual information if possible.

The results presented in Section 5.5 clearly show that the *Flexible External Windowing* scheme outperforms the alternatives with respect to these properties. We have therefore chosen to use this method for the agents of the *UvA Trilearn 2001* soccer simulation team. It has been integrated into the agent's main loop as follows. Each time when the agent receives a message from the server his world model is completely updated. In case of a sense message, the update is actually a prediction since it is based on past observations only. When a see message arrives however, the previous world state is updated according to the new visual information. After updating his world model, the agent chooses an action using the new world state information. If the action is based on visual information from the current cycle it is sent to the server immediately. Otherwise, the action is stored and sent upon the arrival of a signal (if it has not been overwritten; see Algorithm 5.5).

In order to enhance the performance of this method during a real match, we have extended the implementation in two ways. Firstly, we use the action counters included in a sense message (see Section 3.2.3) to avoid clashes. For each type of action command a sense message contains a counter which denotes the number of commands of that type that have been executed by the server. When a new sense message arrives it is therefore possible to check whether the command sent in the previous cycle has been executed by comparing the appropriate counter value to the value in the previous sense message. If the counter has not been incremented we can be sure that the command did not reach the server in time and will thus be executed at the end of the *current* cycle. In this case, we record the previous cycle as a hole and avoid a possible clash in the current cycle by not sending an additional action command to the server. A second extension is that we have enabled our agents to adaptively lower the 90ms send time if a large number of holes occur. This can be caused, for example, by heavy network traffic or the fact that the system is slower than the one used for generating the results in Table 5.1. In our implementation we have made sure that each agent sends an action to the server in every cycle even when he has nothing to do (in this case a **turn** of 0 degrees is 'performed'). It is then possible for the agents to keep track of the number of holes themselves by examining the action counters in each sense message. During a match, the agents check at regular intervals (400 cycles) whether the percentage of holes exceeds 1% (normally it should be less than 0.3%; see Table 5.2) and if this is the case then the send time is lowered. Note that the hole percentage is calculated from scratch for each separate interval (i.e. the calculation is not cumulative).

An additional benefit of sending an action to the server in each cycle is that it enables you to extract the number of holes and clashes from the server logfile in a meaningful way. Clearly, this information would be meaningless if a player would not send an action on each possible occasion, since the occurrence of a hole would not necessarily be caused by a synchronization failure. With the current policy however, parsing the server logfile can thus give an indication of the performance of the synchronization method used. Note that during a real match it is not possible to determine which commands were optimized, since the type of the command sent to the server will depend on the game situation and not on the arrival times of visual messages. Furthermore, it is also not possible to find out which commands reached the server in the correct cycle, since the arguments supplied to each command will no longer be equal to the cycle number from which the command originated. It can thus only be determined in which cycle a command reached the server and not when it was sent. The numbers of holes and clashes are therefore the only statistics that can be derived. Table 5.3 shows these statistics for the entire *UvA Trilearn 2001* soccer simulation team (consisting of 11 players) for a full-length match lasting 6,000 cycles. The numbers on the left correspond to a match which was played using the same three machines (AMD Athlon 700MHz/512MB) that were used for generating the results in Table 5.2, whereas the numbers on the right were generated by playing

	700MHz		**1GHz**	
total cycles	66000	100.00%	66000	100.00%
holes	194	0.29%	13	0.02%
clashes	11	0.02%	0	0.00%

Table 5.3: Synchronization statistics for *UvA Trilearn 2001* for two matches on different systems.

a match on three faster machines (Pentium III 1GHz/256MB). As expected, the results show that on the 700MHz machines the hole percentage for the entire team is almost equal to that in Table 5.2 for the same method. On the faster machines however, holes hardly ever occur anymore and the holes that do occur are never followed by a clash due to the fact that no command is sent to the server if a hole is observed in the previous cycle. From these results it can be concluded that the *Flexible External Windowing* scheme provides a virtually optimal synchronization between an agent and the *soccer server* simulation environment. Note that comparing these statistics to those for other teams would make no sense, since players from other teams do not send an action to the server in each cycle (e.g. to save stamina) which makes the statistics useless for measuring synchronization performance.

Throughout this chapter we have consistently ignored the possibility of trading off the frequency of visual messages against the quality of the given information and the width of the player's view cone (see Equation 3.1). The pseudo-code implementation for the *Flexible External Windowing* scheme presented in Algorithm 5.5 was therefore based on the assumption that the arrival times of see messages describe a pattern that is repeated every three cycles (see Equation 5.2). However, the implementation of the *UvA Trilearn 2001* agents is such that they change the width of their view cone in certain situations which will cause this pattern to change. Note that Equation 5.1 is an instance of the following more general formula that can be used to describe the new pattern:

$$\delta \quad = \quad (t_{see,1} - t_{sense,1}) \ \mathrm{mod} \ gcd(\texttt{send\_step}, \texttt{simulator\_step}) \tag{5.3}$$

where *gcd* is a function that returns the *greatest common divisor* of its two arguments and where `send_step` and `simulator_step` denote the length of the interval between consecutive see messages and the length of a cycle respectively. It is fairly straightforward to extend the synchronization method appropriately for the different cases and we will therefore conclude this chapter by giving an overview of the possible patterns[8]. These patterns can then be used to determine the correct send times for each cycle.

- When the view cone is **narrow** (45 degrees) visual information will come at 75ms intervals and it will thus hold that $\delta \leq 25$. The repeating pattern then becomes $\{\delta, \delta + 75, \delta + 50, \delta + 25\}$.
- When the view cone is **normal** (90 degrees) visual information will come at 150ms intervals and it will thus hold that $\delta \leq 50$. The repeating pattern then becomes $\{\delta, \delta + 50, -\}$ where '$-$' denotes that no visual information arrives during a cycle.
- When the view cone is **wide** (180 degrees) visual information will come at 300ms intervals and it will thus hold that $\delta \leq 100$. The repeating pattern then becomes $\{\delta, -, -\}$.

[8]Here we ignore the possibility of changing the view quality to **low**. The reason for this is that it never happens in our implementation due to the limited usefulness of visual information that contains no distances to objects.

Chapter 6

Agent World Model

In order for an agent to behave intelligently it is important that he keeps a world model that describes the current state of the environment. The agent can then use this world model to reason about the best possible action in a given situation. In this chapter we describe the world model of the agents of the *UvA Trilearn 2001* soccer simulation team. This model can be seen as a probabilistic representation of the world state based on past perceptions. It contains various kinds of information about all the objects on the soccer field as well as several methods which use this information to derive higher-level conclusions. The chapter is organized as follows. Section 6.1 provides a general introduction and discusses several issues which are related to world modeling. The various attributes which are stored in the world model of the agents are described in Section 6.2. Sections $6.3 - 6.6$ are devoted to a discussion of different types of methods which can be used to deal with the world state information in different ways.

6.1 Introduction

A common feature of all agents is that they perceive their environment through sensors and act upon it through actuators. In its simplest form, an agent can be implemented as a direct mapping from perceptions to actions. This means that the agent will respond immediately to perceptual input by associating it with a certain action. A more sophisticated agent however, will not base his actions directly on raw sensory data, but rather on an interpretation of these data. The interpretation of sensory data requires that they are stored using some kind of internal representation and processed using an internal model that reflects the current state of the world. The agent can then use this world model to reason about the best possible action in a given situation. As such, the world model can be seen as one of the most important parts of the agent architecture. It consists of the world as currently perceived by the agent and can be updated both as a result of processed sensory information and according to predicted effects of actions. Since action selection is based on this model, it is clear that it should represent the real world as accurately as possible. Furthermore, the world model should provide an abstraction of the attributes contained in it by supplying methods that enable the agent to reason about the information on a higher level. These methods can be seen to form the basis of the agent's reasoning process.

A robotic soccer agent must be aware of his own position and velocity as well as the positions and velocities of the ball and other players. The agent therefore has to keep a world model that contains this kind of information about various objects. In the *soccer server* each agent has three different types of sensors which

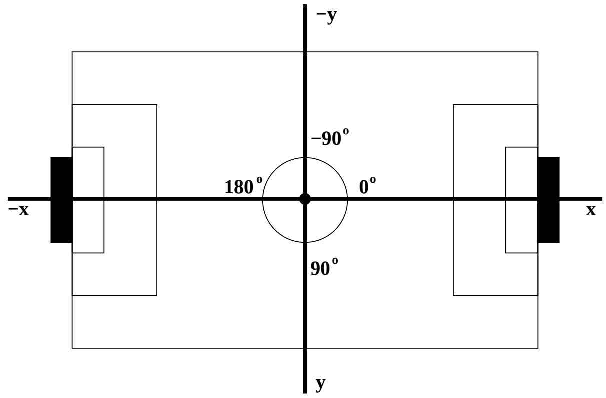


Figure 6.1: Field coordinate system assuming that the opponent's goal is on the right.

provide information about the current world state: a body sensor, a visual sensor and an aural sensor. Information from these sensors is sent to the agents in the form of messages which can be used to update the world model. Besides sensory information, each message also contains a time stamp that indicates the time from which the information originated. By storing this time stamp next to the corresponding information, the agent can distinguish up-to-date world model information from older information. An important aspect of the *soccer server* is that the given sensory information is always relative from the agent's perspective. As a result, an agent cannot directly observe his own global position or the global positions of other objects. The relative information must be converted into global information however, since old relative information is useless once the agent has moved to another position on the field.

The agents of the *UvA Trilearn* soccer simulation team keep a world model that contains information about all the objects on the soccer field. This world model can be seen as a probabilistic representation of the real world based on past perceptions. For each object an estimation of its global position and velocity are stored (among other things) together with a confidence value that indicates the reliability of the estimate. This confidence value is derived from the time stamp which is included in the sensory messages, i.e. if the estimate is based on up-to-date information the associated confidence value will be high. The world model is updated as soon as new sensory information is received by the agent and thus always contains the last known information about the state of the world. Objects which are not visible are also updated based on their last observed velocity. The confidence in the estimate decreases however, for each cycle in which the object is not seen. Furthermore, the agents also use communication to increase the amount of up-to-date information in their world model. As a result, their ability to cooperate with other teammates is greatly enhanced.

As mentioned, all visual information is relative from the agent's perspective and must be converted into global information. This is done by making use of fixed objects (flags, lines and goals) which have been placed on and around the field (see Figure 3.2). Since the global positions of these landmarks are known and do not change, the agent can use them to localize himself by combining the landmark positions with sensory data containing relative information about these objects. Subsequently, the agent's global position can be used to determine the global positions of other objects given their relative positions. The global coordinate system for expressing positions and angles in the agent's world model is shown in Figure 6.1. The center of the field has been chosen as the origin of the system with the x- and y-axes running along the length of the field (i.e. through the centers of both goals) and along the center line respectively. The coordinate system is such that the negative x-direction for a team is towards the goal it defends, whereas the negative y-direction is towards the left side of the field when facing the opponent's goal. Furthermore, global angles are specified in the range $[-180, 180]$. Here the zero-angle points towards the center of the opponent's goal and increases in a clockwise direction. Note that global positions and angles are thus specified in terms of the opponent's side of the field, i.e. independent of whether a team plays from left to right or right to left. This makes it possible to reason about positions and angles in a universal way.

In order to reason about the best possible action, the world model should also provide various methods which enable the agent to use the information on a higher level. The *UvA Trilearn* agent world model contains four types of methods which deal with the world state information in different ways:

- *Retrieval* methods can be used for directly retrieving information about objects.
- *Update* methods can be used to update the world model based on new sensory information.
- *Prediction* methods can be used for predicting future world states based on past perceptions.
- *High-level* methods can be used to derive high-level conclusions from basic world state information.

In the remainder of this chapter, each of these types of methods will be discussed in a separate section. At first however, we will describe the various attributes which are stored in the world model of the agents.

6.2 Attributes

Attributes can be seen as the building blocks of the world model. Each piece of information that is contained in the model is represented by a separate attribute. The attributes thus contain information on which the agent's reasoning process is based. It is therefore of great importance that the attribute values represent the current state of the world as accurately as possible. Each time when the agent receives a new sensory observation the attribute values are updated. In some cases the values can be retrieved directly from the new message, but in most cases the contents of the message first need to be processed before a particular attribute value can be determined. The *UvA Trilearn* agent world model contains a large number of different attributes which can be divided into four main categories: environmental information, match information, object information and action information. A separate subsection will be devoted to each of these categories in which the various attributes that belong to that category will be described.

6.2.1 Environmental Information

The world model holds a number of attributes that contain information which is specific for the environment provided by the *soccer server*. These attributes represent the various server parameters and parameters for heterogeneous player types. When the player client makes a connection with the server he receives three types of messages which are used to assign values to these attributes:

- The first message contains the values for all the server parameters. These parameters specify the behavior of the *soccer server* and define the dynamics of the simulation environment. Examples are `simulator_step` and `ball_speed_max` which respectively denote the duration of a cycle and the maximum speed of the ball. As soon as this message arrives from the server, the values are stored in a separate class called *ServerSettings*. They can be used during the match to perform calculations that predict the outcome of specific actions. Note that the initial values for player-related server parameters are those which apply to the default player type.

- The second message contains the values for several parameters which define the value ranges for heterogeneous player parameters. These values are also stored in the *ServerSettings* class.
- For each heterogeneous player type, the server then sends a message containing the player parameter values for that specific type. These values are randomly chosen from different intervals which are defined by the values in the second message. The intervals are such that the abilities of each player type are based on certain trade-offs (see Section 3.5). Heterogeneous players, for example, are usually faster than default players, but also get tired more quickly. For each player type the corresponding values are stored in a *HeteroPlayerSettings* object. When the coach puts a heterogeneous player of a certain type on the field, this player gets the abilities which are defined by the parameter values for that type. In this case, the player-related parameters in the *ServerSettings* class are updated for this player so that the changed influence of his actions on the environment can be taken into account.

6.2.2 Match Information

The world model also holds several attributes that contain general information about the current state of a match. These are the following:

- *Time*. This attribute represents the server time, i.e. the current cycle in the match. It is represented by an ordered pair (t, s) where t denotes the current server cycle and s is the number of cycles since the clock has stopped. Here the value for t equals that of the time stamp contained in the last message received from the server, whereas the value for s will always be 0 while the game is in progress. It is only during certain dead ball situations (e.g. an offside call leading to a free kick) that this value will be different, since in these cases the server time will stop while cycles continue to pass (i.e. actions can still be performed). Representing the time in this way has the advantage that it allows the players to reason about the number of cycles between events in a meaningful way.
- *PlayerNumber*. This attribute represents the uniform number of the player. Its value is supplied by the server and remains fixed throughout the match. The number can be used to identify players.
- *Side*. This attribute represents the side from which the player's team is playing. Its value is either *left* or *right* and remains fixed throughout the match (even when the teams change sides at half time). The value for this attribute is very important due to the fact that the positions of landmarks are fixed even when the player's coordinate system changes. This thus means that if the team is playing from left to right the global position of a landmark will be different than if it plays from right to left.
- *TeamName*. This attribute represents the name of the player's team and can have any string as its value. It is used when parsing messages from the server to check whether the information is about a teammate or an opponent.
- *PlayMode*. This attribute represents the current play mode and has one of the play modes discussed in Section 3.6 as its value. In general, the play mode equals `play_on` while the game is in progress but will change as a result of certain dead ball situations such as free kicks or corner kicks.
- *GoalDiff*. This attribute represents the goal difference between the teams. Its value is zero when the scores are level, becomes positive when the agent's team is leading and negative when it is trailing.

6.2.3 Object Information

The world model contains information about all the objects on the soccer field. These objects can be divided into *stationary* objects which have a fixed position (flags, lines and goals) and *dynamic* objects which move around (players and the ball). For each object, the information is stored in the *Object* class which serves as a container for the data and provides no additional functionality other than methods for setting and getting the information. Positions and velocities are stored in global coordinates since these are independent of the agent's position on the field, i.e. unlike relative coordinates they do not change when the agent moves around. The agent uses the stationary objects (landmarks) on the field to localize himself given relative perceptions about these objects. Based on his own global position he is then able to translate the relative information about other objects into global information as well. Figure 6.2 shows an UML class diagram for the object type hierarchy (refer to Section 4.3 for details about UML class diagrams). Note that for each class only the most important attributes are shown and not the auxiliary ones which are necessary for calculating these values. The relative positions of objects as obtained from server messages, for example, are omitted. Furthermore, the operations for each class have also been omitted for space reasons and because they provide no functionality other than setting or getting the attribute values. The information contained in each class is described in more detail below.

- *Object*. This is the abstract superclass of the object type hierarchy which contains basic information about an object. Its most important attributes denote the type of the object, the global position of the object and the time at which the object was last observed. Note that the 'object type' attribute uniquely defines the identity of the object (e.g. `OBJECT_BALL`, `OBJECT_GOAL_R`, `OBJECT_TEAMMATE_9`, `OBJECT_OPPONENT_6`, etc.). Furthermore, the 'time last seen' attribute can be used to derive a confidence value that indicates the reliability of the object information [90]. This confidence value comes from the interval $[0, 1]$ and is defined by the following formula:

  $$confidence \quad = \quad 1 - (t_c - t_o)/100 \qquad (6.1)$$

  where t_c denotes the current time and t_o the time at which the object was last observed. This means that the confidence decreases by 0.01 for every cycle in which the object is not seen and is restored to 1.0 when the object is observed again. Note that the confidence value has been introduced because the agent only has a partial view of the environment at any moment. If an object has not been in view for a number of cycles, its current position is estimated based on its last observed velocity. However, the estimate will become less reliable as this number of cycles increases due to the fact that the agent does not know which actions the object has performed[1]. As such, the confidence value is thus an important reliability measure for object information.

- *FixedObject*. This is a subclass of the *Object* class which contains information about the stationary objects on the field. It adds no additional attributes to those inherited from the *Object* superclass.

- *DynamicObject*. This is also a subclass of the *Object* class which contains information about mobile objects. It adds velocity information to the general information provided by the *Object* class. The most important additional attributes denote the global velocity of the object and the time at which this velocity was last observed.

- *PlayerObject*. This is a subclass of the *DynamicObject* class which contains information about a specific player on the field (either a teammate or an opponent). It adds attributes denoting the global neck angle and global body angle of the player as well as the time at which these angles were last observed to the information provided by the *DynamicObject* class. Furthermore, it holds a boolean attribute which indicates whether the player is a goalkeeper or not. Note that the agent himself is not a member of this class.

[1] Note that the confidence value for landmarks will always be 1.0 due to the fact that landmarks have a fixed global position.

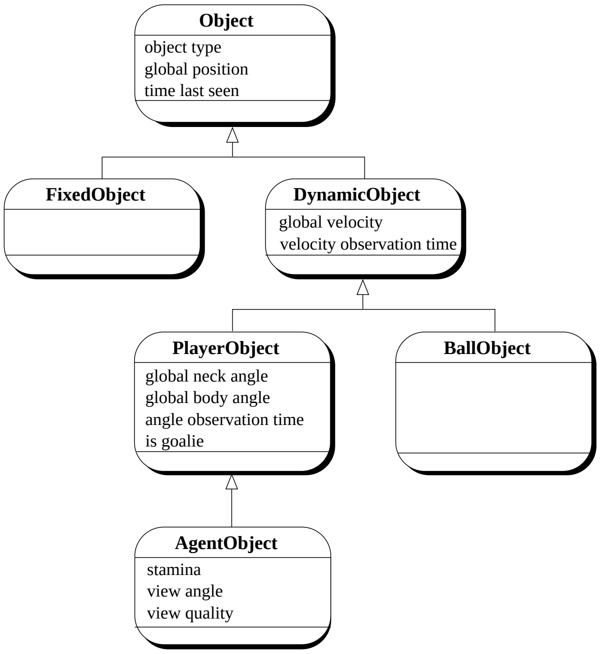


Figure 6.2: UML class diagram of the object type hierarchy.

- *BallObject*. This is also a subclass of the *DynamicObject* class which contains information about the ball. It adds no additional attributes to those inherited from the *DynamicObject* superclass.
- *AgentObject*. This is a subclass of the *PlayerObject* class which contains information about the agent himself. It adds attributes denoting the stamina, view angle and view quality of the agent to the information provided by the *PlayerObject* class.

The world model contains information about all the visible flags as well as the furthest visible line. The visual sensor perceives a line when the bisector of the agent's view cone intersects this line. This means that when the agent is located inside the field he will always see only a single line. In cases where the agent is positioned outside the field (e.g. to perform a 'kick in'), we only store information about the furthest visible line because the orientation to this line is the same as when the agent would be standing inside the field facing in the same direction. Furthermore, the world model contains information about all the dynamic objects on the field: the ball, 11 opponent players, 10 teammates and the agent himself. Information about players is stored based on their uniform number, i.e. the uniform number can be used to index the player information. The biggest challenge for keeping track of all the players is that if the distance to a player is very large his uniform number and possibly team name will not be visible (see Section 3.2.1). Visual information received by the agent thus often will not identify a player that is seen. In these cases, previous player positions are used to help disambiguate the identity. By combining the position and velocity information of a player currently stored in the world model, it is possible to determine whether an observed player could be the same as this previously identified one. If there is a possible match, it is assumed that the observed player is indeed the same as the one for which information was already available. When no match is found however, the player information is linked to a player for which the confidence value has dropped below a certain threshold. Note that this 'anonymous' player information can still be very useful since the evaluation of passing options, for example, only requires knowledge about the presence of a player and not about his uniform number.

6.2.4 Action Information

The world model also holds several attributes that contain information about the actions that the agent has previously performed. These are the following:

- *NrKicks*, *NrDashes*, *NrTurns*, *NrSays*, *NrTurnNecks*, *NrCatches*, *NrMoves*, *NrChangeViews*. These attributes are counters for the total number of actions of a certain type which have been executed by the server; e.g. when *NrTurns* equals 38 this means that the agent has so far executed 38 **turn** commands. The values for these attributes are directly obtained from the information included in **sense_body** messages (see Section 3.2.3). The counters are used to determine whether an action command sent in the previous cycle has been performed. If the appropriate counter has not been incremented we can be sure that the command did not reach the server before the end of the cycle and will thus be executed in the *current* cycle instead of the previous one.
- *QueuedActions*. This attribute represents a list that contains all the actions that the agent has sent to the server in the previous cycle.
- *PerformedActions*. This attribute represents a list of booleans that indicate for each action in the *QueuedActions* list whether this action has been performed. The booleans are set by making use of the counter values. The *PerformedActions* attribute thus represents the agent's knowledge about which actions have been performed and can be used to update the world model accordingly.

6.3 Retrieval Methods

Retrieval methods can be used for directly retrieving information about objects from the agent's world model. For each of the attributes described in Section 6.2, a `getAttributeName` method has been defined which can be used to get the corresponding value. For attributes representing environmental information, match information or action information these methods can be called without any arguments. The method call `getTime()`, for example, will return the current time. For attributes representing object information this is not possible however, since these attributes are defined for multiple objects. To obtain an attribute value for a specific object one must thus supply an object identifier as an argument to the corresponding retrieval method. This identifier is then used to look up the information for that particular object. The method call `getGlobalPosition(OBJECT_TEAMMATE_3)`, for example, will return the currently stored global position of the teammate with uniform number three. Methods have also been defined for iterating over a group of objects. This can be useful when information about different objects has to be compared.

6.4 Update Methods

Update methods can be used to update the world model based on new sensory information from the server. For each of the attributes described in Section 6.2, a `setAttributeName` method has been defined which can be used to set the corresponding value. For some attributes the values can be directly extracted from the information in a sensory message, whereas others need to be calculated based on the new information. The world model is updated each time when new sensory information arrives from the server. Such an update can be divided into two separate phases. The first phase is the *process* phase in which for each object the information contained in the message is stored at the right location in the world model. In the *process* phase a value is thus assigned to each attribute that represents a piece of information that is directly present in the sensory message. The second phase is the *update* phase in which the new values are combined with the information already present in the world model to determine up-to-date values for the remaining attributes. In this way all the world model data are always based on the last known information about the state of the world. Note that an update of the world model will either be based on new visual information or on predictions of future states. In this section we will discuss for each type of sensory message (physical, visual and aural) how the two phases mentioned earlier have been implemented to update the values in the agent's world model.

6.4.1 Update from Body Sensor

The body sensor reports physical information about an agent. At the beginning of each simulation cycle the agent receives a **sense_body** message from the server that contains the following information:

- The time t (i.e. the current cycle) at which the information applies.
- The agent's view angle and view quality.
- The agent's stamina.
- Counters for the number of times that the agent has performed a certain action.
- The velocity $(\tilde{v}_x^t, \tilde{v}_y^t)$ of the agent relative to the direction of his neck.
- The neck angle $\tilde{\theta}^t$ of the agent relative to his body.

As soon as such a message arrives, it is processed by storing the information at the appropriate location in the agent's world model (*process* phase). The agent's speed and neck angle are then used to update his global position and velocity (*update* phase) in an analytical way by making use of the known *soccer server* dynamics. During each simulation cycle, the movement of mobile objects in the *soccer server* is calculated according to the following equations (neglecting motion noise):

$$(u_x^{t+1}, u_y^{t+1}) = (v_x^t, v_y^t) + (a_x^t, a_y^t)\text{: accelerate} \tag{6.2}$$
$$(p_x^{t+1}, p_y^{t+1}) = (p_x^t, p_y^t) + (u_x^{t+1}, u_y^{t+1})\text{: move} \tag{6.3}$$
$$(v_x^{t+1}, v_y^{t+1}) = \mathit{Decay} \times (u_x^{t+1}, u_y^{t+1})\text{: decay speed} \tag{6.4}$$
$$(a_x^{t+1}, a_y^{t+1}) = (0, 0)\text{: reset acceleration} \tag{6.5}$$

Here (p_x^t, p_y^t), (v_x^t, v_y^t) and (a_x^t, a_y^t) respectively denote the position, velocity and acceleration of the object in cycle t. *Decay* is a parameter representing the velocity decay rate of the object. These equations can be used to update the global position and velocity of the agent based on the information in the physical message. Equation 6.3 shows that the difference between the global positions in the previous and current cycles equals the movement (u_x^t, u_y^t). This movement vector can be derived by rewriting Equation 6.4:

$$(u_x^t, u_y^t) = \frac{(v_x^t, v_y^t)}{\mathit{Decay}} \tag{6.6}$$

In order to determine the movement, we thus have to compute the global velocity (v_x^t, v_y^t) of the agent in the current cycle. This can be done by making use of the relative velocity $(\tilde{v}_x^t, \tilde{v}_y^t)$ that is contained in the **sense_body** message. The key observation is that (v_x^t, v_y^t) and $(\tilde{v}_x^t, \tilde{v}_y^t)$ by definition will have the same magnitude (representing speed). However, their directions will be different. The global velocity can thus be determined by performing a simple rotation of the relative velocity vector. Since $(\tilde{v}_x^t, \tilde{v}_y^t)$ denotes the velocity of the agent relative to the direction of his neck, the rotation angle in this case will be equal to the agent's global neck angle θ^t. This angle can be determined by updating the global neck angle θ^{t-1} (which is stored in the world model) based on the actions that the agent has sent to the server in the previous cycle. Note that we first have to check whether these actions have been performed by comparing the action counters included in the current sense message to the corresponding values in the previous one. For determining θ^t we are only interested in **turn** and **turn_neck** commands since these will change the agent's global neck direction if they are performed. The command parameters for these two actions can be used to update the global neck angle in the following way:

$$\theta^t = \mathit{normalize}(\theta^{t-1} + \alpha + \nu) \tag{6.7}$$

where α and ν respectively denote the actual angles by which the agent has turned his body and his neck in the previous cycle and where *normalize* is a function that converts the resulting angle to an equivalent angle from the interval $[-180, 180]$. Recall from Section 3.4.3 that the actual angle by which a player turns his body is usually not equal to the *Moment* argument supplied to the **turn** command. Instead, the actual turn angle depends on the speed of the player, i.e. as the player moves faster it is more difficult for him to turn due to his inertia. Since a player cannot execute a **dash** and a **turn** in the same cycle, his acceleration (a_x^t, a_y^t) will become zero when a **turn** is performed. As a result, an expression for the global velocity (v_x^{t-1}, v_y^{t-1}) in the previous cycle can be derived by combining (6.2) and (6.4):

$$(v_x^{t-1}, v_y^{t-1}) = \frac{(v_x^t, v_y^t)}{\mathit{Decay}} \tag{6.8}$$

Since the **turn** has been performed in the previous cycle, the actual turn angle only depends on the speed in that cycle. This speed can be determined by inversely applying the agent's speed decay to his global velocity in the current cycle, i.e. by replacing the vectors in Equation 6.8 by their norms. The actual turn angle α as defined by Equation 3.31 then becomes:

$$\alpha = \frac{(1.0 + \tilde{r}) \cdot \mathit{Moment}}{(1.0 + \texttt{inertia\_moment}) \cdot \|(v_x^{t-1}, v_y^{t-1})\|} \tag{6.9}$$

where *Moment* denotes the argument supplied to the **turn** command, `inertia_moment` is a server parameter representing a player's inertia and $\tilde{r}$ is a random number taken from a uniform distribution over the [−`player_rand`, `player_rand`] interval. Since we have no past observations to filter the noise factor $\tilde{r}$, the best prediction for α will be at the mean of the uniform noise distribution. The noise in Equation 6.9 is therefore neglected when computing the actual turn angle. The resulting value for α and the actual angle ν by which the agent has turned his neck (and which by definition equals the argument supplied to the **turn_neck** command) can now be used to determine the agent's global neck angle θ^t in the current cycle using Equation 6.7. Rotating the relative velocity $(\tilde{v}_x^t, \tilde{v}_y^t)$ over θ^t degrees then gives the global velocity (v_x^t, v_y^t) in the current cycle after which (6.6) and (6.3) can be used to update the agent's global position.

Note that in the above derivation we do not take previous velocity perceptions into account to improve our global velocity estimation. The old perceptions are neglected for two reasons. Firstly, the noise added to the motion of an object is propagated meaning that it actually changes the direction of the velocity vector. The noise is thus actually affecting the movement and is not just included into the sensory data. Due to this noise propagation, the currently perceived velocity will be based on a different direction of movement than the velocity in the previous cycle. Secondly, the amount of sensor noise in the perceived velocity is only small, whereas the amount of motion noise that is propagated from the previous cycle is rather large (see Sections 3.2.3 and 3.3). The old perception therefore has little value for the prediction.

When a sense message arrives from the server, the global positions and velocities of other players and the ball are also updated. Due to the fact that no information is available about which actions are performed by other players, no assumptions are made about their behaviour in the previous cycle. This means that their positions and velocities are updated according to Equations 6.2 to 6.5 with zero acceleration. In order to update the information about the ball, it is checked whether the agent has performed a **kick** in the previous cycle. This is again done by comparing the counter values for the **kick** command in subsequent sense messages. If the agent has indeed executed a **kick** and if the ball was within his kickable distance at the time, it is assumed that the ball has been accelerated as a result of the **kick**. In this case the ball's global position and velocity are updated based on Equations 6.2 to 6.5 with

$$(a_x^{t-1}, a_y^{t-1}) \quad = \quad act\_pow \times \texttt{kick\_power\_rate} \times (cos(\theta_{ball}^{t-1}), sin(\theta_{ball}^{t-1})) \qquad (6.10)$$

where *act_pow* is the actual kick power as defined by Equation 3.26, `kick_power_rate` is a server parameter which is used to determine the size of the acceleration vector and θ_{ball}^{t-1} is the direction in which the ball is accelerated in cycle $t-1$. This direction is equal to the sum of the agent's body direction in the previous cycle and the *Angle* parameter supplied to the **kick** command. If the agent did not perform a **kick** in the previous cycle, it is assumed that the ball has not been kicked at all[2]. In this case, the ball information is updated according to Equations 6.2−6.5 with zero acceleration. If the positions of the ball and the agent overlap after the update, these positions are adapted according to the known *soccer server* collision model. This means that both objects are moved back into the direction where they came from until they do not overlap anymore. After this, their velocities are multiplied by -0.1.

Note that in order to update the agent's world model based on a **sense_body** message several unreliable assumptions have to be made due to a lack of information. Examples are that it is assumed that the ball is never kicked by other players and that other players do not perform actions. Since the arrival of a **sense_body** message indicates the start of a new cycle however, this uncertainty is represented by the fact that the confidence values associated with the world model information have dropped according to Equation 6.1. It is only when the agent receives visual information about an object that the confidence in the object information is restored to its maximum value.

6.4.2 Update from Visual Sensor

The visual sensor detects visual information about objects in the agent's current field of view. This information is automatically sent to the agent every `send_step` (in the current server version 150) ms in the form of a **see** message that contains the following information:

- The time t (i.e. the current cycle) at which the information applies.
- Information about the visible objects on the field. This object information is relative from the agent's perspective (i.e. relative to his neck angle) and contains the following elements:
  - The name by which the object can be uniquely identified.
  - The distance r to the object.
  - The direction ϕ to the object.
  - The distance change Δr of the object.
  - The direction change $\Delta\phi$ of the object.
  - The body direction $\tilde{\theta}_{body}^t$ of the object.
  - The neck direction $\tilde{\theta}_{neck}^t$ of the object.

[2]Assumptions about other players executing kicks cannot be made in a meaningful way.

For stationary objects only the name, distance and direction are given. For dynamic objects the amount of information given depends on the distance to the object. The name, distance and direction to these objects is always included, whereas the probability of receiving the last four items listed above decreases as the distance to an object increases (see Section 3.2.1).

The world model is updated as soon as such a message arrives. During the *process* phase the information contained in the **see** message is directly stored at the right location in the world model. After this, the information about the agent himself is updated first. The agent's global position is determined by making use of the visible landmarks on and around the field. This global position is then used to update the positions and velocities of other dynamic objects. Agent localization is thus extremely important since errors in the self-localization method will propagate to all object information. In this section we compare several methods for determining the agent's global position and orientation using the available landmarks. We then describe a number of methods for updating the positions and velocities of other dynamic objects based on the new information. Note that the global velocity of the agent himself can only be calculated from the information in a visual message by looking at the difference between the agent's new global position and his previous one. It is not necessary to do this however, since the global velocity can be derived more accurately from the information received from the body sensor as described in Section 6.4.1.

6.4.2.1 Agent Localization

The various landmarks (flags and lines) which have been placed on and around the field are shown in Figure 3.2. These landmarks can be used for self-localization. The agent will perceive a different number of landmarks depending on his neck direction and location on the field. For example, when facing the center of the field a large number of landmarks will be visible, whereas this number will become much smaller when standing in a corner looking towards the closest side line. Clearly, a good localization method using landmarks needs to have the following important properties:

- It must be *accurate*, i.e. the error in the resulting position estimate must be as small as possible.
- It must be *robust*, i.e. it should still work when only a small number of landmarks are visible.
- It must be *fast*, i.e. computationally cheap to satisfy the agent's real-time demands.

In this section we describe three methods which can be used for agent localization in the *soccer server*. These methods were experimentally compared based on a number of possible configurations for determining the agent's global position. The results of this experiment are presented at the end of this section.

Method 1: agent localization using one line and one flag

The first method uses the relative visual information about one line and one flag to determine the agent's global position. The direction to this line can be used to calculate the agent's global neck angle and subsequently his global position by making use of the fixed global position of the flag which is known to the agent. Let l_b be the bisector of the agent's view cone, i.e. the half line starting at the agent's global position which extends into the direction of his neck. When the agent sees a line l, the reported visual information consists of the distance d to the intersection point of l and l_b and the angle α between l and l_b. Figure 6.3 shows an example situation. Note that the reported angle α equals the acute angle by which l_b has to be rotated around the intersection point to be aligned with l. This angle will be positive in case of a clockwise rotation and negative otherwise (see Figure 6.1). In order to obtain the agent's global neck

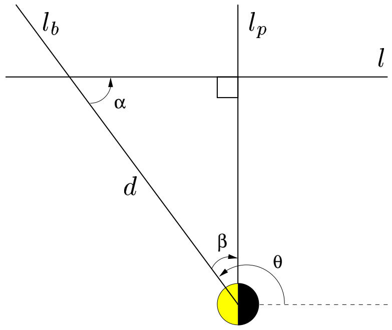


Figure 6.3: Determining the agent's global neck angle θ using the reported angle α to a line l.

angle θ, we need to determine the angle β between l_b and the perpendicular line l_p depicted in Figure 6.3. The angle β can be calculated according to the following formula:

$$\beta = -sign(\alpha)(90 - |\alpha|) \tag{6.11}$$

where *sign* is a function that returns +1 if its argument is positive and −1 otherwise. Note that it will always be the case that $sign(\beta) = -sign(\alpha)$ because the orientation of the agent's neck relative to the line l will always be the reverse of the orientation relative to the perpendicular line l_p. In order to determine the global neck angle θ of the agent we also need to know the global orientation of the perpendicular line l_p. For each possible l_p this orientation can be easily derived using Figure 6.1. The global orientations of the lines perpendicular to each of the four side lines (top, bottom, left and right) are shown in Table 6.1 for the team that plays from left to right. For the opposite team the opponent's goal is located on the other side of the field and 180 degrees thus have to be added to each orientation.

Side line	Perpendicular orientation
`OBJECT_LINE_R`	0
`OBJECT_LINE_B`	90
`OBJECT_LINE_L`	180
`OBJECT_LINE_T`	-90

Table 6.1: Global orientations (in degrees) of lines perpendicular to each of the four side lines for the left team. For the right team 180 degrees have to be added to each orientation.

Note that we assume here that a side line is perceived from a location inside the field[3]. Using the values in Table 6.1 the agent's global neck angle θ can be calculated as follows:

$$\theta = \text{orientation}(l_p) - \beta \tag{6.12}$$

We can now use the position of the flag relative to the agent to determine his global position on the field. Visual information about a flag f consists of the distance $\tilde{f_r}$ and direction $\tilde{f_\phi}$ to this flag relative to the neck of the agent. The visual message thus effectively contains the relative position $(\tilde{f_r}, \tilde{f_\phi})$ of the flag in polar coordinates. Furthermore, the agent also knows the fixed global position (f_x, f_y) of the flag[4]. By combining the global and relative information about the flag, it is possible to determine the agent's global position (p_x, p_y). To this end, we first have to perform a rotation to align the relative coordinate

[3]This is why we only store information about the furthest visible line when the agent sees two lines from outside the field.
[4]The global positions of all the stationary objects (landmarks) are stored in the agent's world model.

system (polar) with the global coordinate system (Cartesian). Since all visual information is relative to the neck of the agent, the rotation angle will be equal to the agent's global neck direction θ. The agent's global position (p_x, p_y) is then obtained by converting the polar coordinates to Cartesian coordinates and by performing a translation that matches the flag's relative position to its global position. In summary, (p_x, p_y) can thus be calculated as follows:

$$(p_x, p_y) = (f_x, f_y) - \pi(\tilde{f}_r, \tilde{f}_\phi + \theta) \tag{6.13}$$

where π is a function that converts polar coordinates to Cartesian coordinates, i.e.

$$(x, y) = \pi(r, \phi) = (r \cdot \cos(\phi), r \cdot \sin(\phi)) \tag{6.14}$$

Method 2: agent localization using two flags

The second method uses the relative visual information about two flags to determine the global position of the agent. Let the following information be included in a visual message:

- The distance $\tilde{f}_r$ and direction $\tilde{f}_\phi$ to a flag named f.
- The distance $\tilde{g}_r$ and direction $\tilde{g}_\phi$ to a flag named g.

In addition, the agent also knows the global positions (f_x, f_y) and (g_x, g_y) of f and g since these are stored in his world model. Combining the distance to a flag with the known global position of this flag gives a circle of possible agent positions. This circle has the flag as its center and the perceived distance as its radius. In the same way the second flag also defines a circle of possible positions. Clearly, the correct agent position must lie on both circles. In general, the circles will have two intersection points one of which denotes the actual position of the agent. It can be determined which of these two points is the correct one by looking at the difference between the angles at which both flags are perceived.

An example situation is shown in Figure 6.4. Following this approach, the agent localization problem thus amounts to finding the correct intersection point p of the two circles. In order to determine this point, we first calculate the orthogonal projection p' of p onto the line segment that connects the two flags. To this end we compute the distance d between f and g by making use of their known global positions, i.e.

$$d = \sqrt{(g_x - f_x)^2 + (g_y - f_y)^2} \tag{6.15}$$

This distance can be seen to consist of two parts: one from f to p' and one from p' to g. Let a denote the distance from f to p', b the distance from p' to g and h the distance from p to p'. When we consider the triangles *fpp'* and *gpp'* it follows from the Pythagorean theorem that

$$\tilde{f}_r^2 = a^2 + h^2 \tag{6.16}$$

$$\tilde{g}_r^2 = b^2 + h^2 \tag{6.17}$$

These equations and the fact that $b = d - a$ can be used to derive the value for a. This gives:

$$a = \frac{\tilde{f}_r^2 - \tilde{g}_r^2 + d^2}{2d} \tag{6.18}$$

The global coordinates of p' can now be determined as follows:

$$(p'_x, p'_y) = (f_x + a \cdot \cos(\alpha), f_y + a \cdot \sin(\alpha)) \tag{6.19}$$

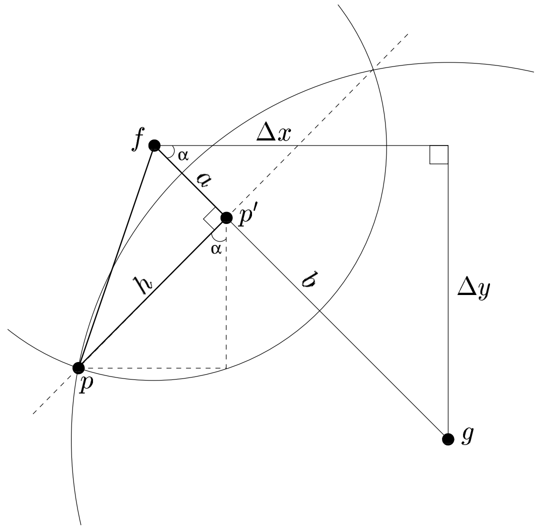


Figure 6.4: Agent localization using two flags (f and g).

Here $\cos(\alpha)$ and $\sin(\alpha)$ are the normalized values for Δx and Δy as depicted in Figure 6.4, i.e. $\cos(\alpha) = \Delta x/d$ and $\sin(\alpha) = \Delta y/d$. The global position p of the agent then becomes:

$$(p_x, p_y) \quad = \quad (p'_x - h \cdot Sign \cdot \sin(\alpha), p'_y + h \cdot Sign \cdot \cos(\alpha)) \tag{6.20}$$

where $h = \sqrt{(\tilde{f}_r^2 - a^2)}$ as can be derived from (6.16) and where $Sign$ equals $+1$ when $\tilde{g}_\phi - \tilde{f}_\phi$ is positive and -1 otherwise. The value for $Sign$ thus determines which of the two intersection points denotes the correct position of the agent. Note that it is also possible to use a third landmark to disambiguate the agent's global position since by definition the resulting third circle will provide a unique intersection point. This has not been done for two reasons however. Firstly, the three circles often do not exactly intersect one another in a common point due to the noise which is included in the sensory observations. Furthermore, using only two landmarks better meets the robustness property mentioned at the beginning of this section.

Method 3: agent localization using a particle filter

The third method makes use of a *particle filter* [28, 110, 117] to determine the global position of the agent. A convenient way to look at the agent localization problem is through a state-space approach. The agent can be seen as a controllable Markov process with hidden (unobserved) low-dimensional states $\mathbf{x}_t \in \mathcal{X} \subset \mathbb{R}^q$ for each time step t. For the current problem these states correspond to positions. We assume an initial state $\mathbf{x}_0$ and an initial distribution $P(\mathbf{x}_0)$ at time $t = 0$. Furthermore, we regard the known *soccer server* dynamics (including noise) as a stochastic transition model $P(\mathbf{x}_{t+1}|\mathbf{x}_t, a_t)$ which brings the agent stochastically from state $\mathbf{x}_t$ to state $\mathbf{x}_{t+1}$ for an action a_t issued at time t. Assuming that sensory observations $\mathbf{y}_t$ sent by the server in each cycle are conditionally independent given the states $\mathbf{x}_t$, the agent localization problem boils down to estimating in each time step t a posterior density $P(\mathbf{x}_t|\mathbf{y}_t)$ over the state space $\mathcal{X}$ that describes the agent's current belief about its state at time t. Using the Bayes rule, the following proportionality can be derived for this posterior:

$$P(\mathbf{x}_{t+1}|\mathbf{y}_{t+1}) \quad \propto \quad P(\mathbf{y}_{t+1}|\mathbf{x}_{t+1})\, P(\mathbf{x}_{t+1}) \tag{6.21}$$

where the prior density $P(\mathbf{x}_{t+1})$ corresponds to the propagated posterior from the previous time step, i.e.

$$P(\mathbf{x}_{t+1}) \quad = \quad \int P(\mathbf{x}_{t+1}|\mathbf{x}_t)\, P(\mathbf{x}_t|\mathbf{y}_t)\, d\mathbf{x}_t \tag{6.22}$$

Here the Markov assumption that the past has no effect beyond the previous time step has been implicitly used. Equations 6.21 and 6.22 provide an efficient iterative scheme for Bayesian filtering. However, in order to determine the posterior analytically, we must be able to compute the integral in (6.22). The result can then be multiplied by the likelihood $P(\mathbf{y}_{t+1}|\mathbf{x}_{t+1})$ after which the resulting density $P(\mathbf{x}_{t+1}|\mathbf{y}_{t+1})$ can be normalized to unit integral. It turns out that the posterior can only be analytically computed when the transition and observation models are linear-Gaussian [117]. Since this is not the case for the *soccer server* simulation environment, we must resort to approximations or simulation.

The particle filter is an attractive simulation-based approach to the problem of computing intractable posterior distributions in Bayesian filtering [28]. The idea is to determine a discrete approximation of the continuous posterior density in (6.21) by using a set of N particles x_t^i with associated probability masses π_t^i for $i = 1, ..., N$. An empirical estimate for the posterior is then given by

$$P(\mathbf{x}_t|\mathbf{y}_t) \quad = \quad \sum_{i=1}^{N} \pi_t^i\, \delta(x_t - x_t^i) \tag{6.23}$$

with $\delta(x_t - x_t^i)$ a delta function centered on the particle x_t^i. Using (6.23), the integration for computing the prior in (6.22) is now replaced by the following summation:

$$P(\mathbf{x}_{t+1}) = \sum_{i=1}^{N} \pi_t^i\, P(\mathbf{x}_{t+1}|\mathbf{x}_t^i) \tag{6.24}$$

Since all the integrals are replaced by sums and all the continuous densities by discrete ones, the required normalization step of the filtered posterior

$$P(\mathbf{x}_{t+1}|\mathbf{y}_{t+1}) \quad \propto \quad P(\mathbf{y}_{t+1}|\mathbf{x}_{t+1}) \sum_{i=1}^{N} \pi_t^i\, P(\mathbf{x}_{t+1}|\mathbf{x}_t^i) \tag{6.25}$$

becomes trivial, namely, a normalization of the discrete masses to unit sum.

For the problem of agent localization in the *soccer server* we only use the visual information received in the current cycle to determine the agent's position. However, each visual message contains multiple pieces of information (i.e. information about several objects) which can be regarded as subsequent observations in the light of the above theory. We have implemented the particle filter method for agent localization as follows. Assume that we have an initial estimate $\mathbf{x}_t$ of the agent's global position (e.g. based on his previous position or using one of the methods described earlier). The posterior density $P(\mathbf{x}_t|\mathbf{y}_t)$ is then approximated by using a grid of particles which is centered on this estimated position. Each particle on the grid has an associated probability mass π^i which is equal to $1/N$ with N the total number of particles. The reason for choosing a uniform distribution is that we have no measure for the quality of the initial estimate. Initially, the probability for each particle is thus the same and each particle can be seen as a 'candidate' position. We can now use the perceived distance and direction to a flag to reduce the number of candidates. Recall from Section 3.2.1 that noise is incorporated into the visual sensor data by quantizing the values sent by the server. Distances to flags are quantized as follows:

$$r' \quad = \quad \text{Quantize}(\exp(\text{Quantize}(\ln(r), \texttt{quantize\_step\_l})), 0.1) \tag{6.26}$$

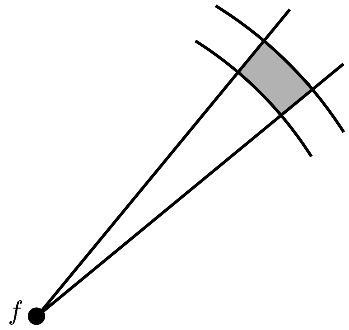


Figure 6.5: The perception of a flag f yields a range of possible agent positions as indicated by the shaded area. This is due to the sensor noise that is incorporated into visual observations.

where r and r' are the exact and quantized distances respectively and where `quantize_step_l` is a server parameter which represents the quantize step. Furthermore,

$$\text{Quantize}(V, Q) \quad = \quad \text{rint}(V/Q) \cdot Q \tag{6.27}$$

where 'rint' denotes a function which rounds a value to the nearest integer. This quantization procedure effectively means that a range of real distances r to a flag is mapped to the same quantized distance r' which is included in the visual message. Given the quantized distance r' we can determine the minimum and maximum value for the real distance r. Note that in Equation 6.26 the second argument supplied to the Quantize formula is always a constant value (either 0.1 or `quantize_step_l` which equals 0.01 in *soccer server* version 7.10). Assuming that $\text{Quantize}(V, Q) = q$, this enables us to calculate the minimum and maximum possible value for the first argument V by 'inverting' Equation 6.27. This gives:

$$V_{min} \quad = \quad \text{invQMin}(q, Q) = (\text{rint}(\frac{q}{Q}) - 0.5) \cdot Q \tag{6.28}$$

$$V_{max} \quad = \quad \text{invQMax}(q, Q) = (\text{rint}(\frac{q}{Q}) + 0.5) \cdot Q \tag{6.29}$$

By applying this twice in (6.26) we get the range from which the real distance r must originate:

$$r_{min} \quad = \quad \exp(\text{invQMin}(\ln(\text{invQMin}(r', 0.1)), \texttt{quantize\_step\_l})) \tag{6.30}$$

$$r_{max} \quad = \quad \exp(\text{invQMax}(\ln(\text{invQMax}(r', 0.1)), \texttt{quantize\_step\_l})) \tag{6.31}$$

Using the perceived distance r' we can update the particle grid by removing all the particles that do not fall within the computed range $[r_{min}, r_{max}]$ of real distances. We can then further update the particle grid by taking into account the perceived direction ϕ' to the flag. Directions to flags and lines are quantized by rounding the real value to the nearest integer (see Equation 3.18). This means that the quantized direction can deviate at most 0.5 degrees from the real value. However, the same holds for the agent's global neck angle θ which is also estimated based on a directional observation (see Section 6.4.2.2). Given the quantized direction ϕ', the range from which the real direction ϕ must originate thus equals

$$[(\phi' - 1), (\phi' + 1)] \tag{6.32}$$

Based on this information we can again remove all the particles for which the perceived direction to the flag (given the agent's global neck angle) falls outside the range in (6.32). An example of how the perception of a flag yields a range of possible agent positions is shown in Figure 6.5. After removing the 'illegal' particles from the grid using this range information, the original number of particles is restored by resampling the grid in the 'legal' area (i.e. from the posterior in (6.25)). This is done by randomly selecting one of the remaining particles and centering a Gaussian kernel on this point for which the standard deviation σ reflects the local density. Following [118], we choose σ as follows:

$$\sigma = \frac{1}{12}\sqrt{\frac{1}{n}\sum_{i=1}^{n}\|\mathbf{x}_t^i - \bar{\mathbf{x}}\|^2}\left(\frac{4}{n(d+2)}\right)^{\frac{1}{d+4}} \tag{6.33}$$

with n the number of remaining particles, $\bar{\mathbf{x}}$ the mean of all remaining particles and d the dimension of the data (in our case $d = 2$ since the particles denote positions). This σ has the property that it minimizes the mean integrated squared error between a hypothetical Gaussian posterior distribution and the resampled particle set. A new particle is then created by drawing a sample from the Gaussian. This is done for randomly selected particles until the original number of samples has been restored. The particle filter procedure is repeated for every observation $\mathbf{y}_t$. This means that the particle grid is updated for each visible flag after which the grid is again resampled. When all the flags have been processed we determine the average of the remaining set of particles to obtain our global position estimate.

Results

We have compared the performances of the three methods described above by conducting an agent localization experiment. For this experiment we used a number of different configurations:

- *Configuration 1.* The first method is applied to localize the agent using the furthest visible line and the closest visible flag.
- *Configuration 2.* The first method is applied to localize the agent using the furthest visible line and all the visible flags. For each flag the visual information is combined with the information about the line to obtain a global position estimate. This estimate is then weighted depending on the distance to the flag. We have already seen that, given the perceived distance r' to a flag, it is possible to determine the range of values $[r_{min}, r_{max}]$ from which the real distance r must originate by inverting (6.26). Each new measurement y_{i+1} is now weighted according to the variance σ_{i+1}^2 of this possible range of values. Since the values in this range are uniformly distributed this gives:

$$\sigma_{i+1}^2 = \frac{(r_{max} - r_{min})^2}{12} \tag{6.34}$$

  Using a simple recursive parameter estimation method (Kalman filter [43]) the current estimate is updated by taking into account the new measurement with its associated variance. For each new measurement y_{i+1}, the current estimate $\hat{x}_i$ is corrected by adding a weighted difference as follows:

$$\hat{x}_{i+1} = \hat{x}_i + K \cdot (y_{i+1} - \hat{x}_i) \tag{6.35}$$

  where the correction term K equals

$$K = \frac{\hat{\sigma}_i^2}{\hat{\sigma}_i^2 + \sigma_{i+1}^2} \tag{6.36}$$

  This means that the sample weight will be smaller if the distance (and consequently σ_{i+1}^2) to a perceived flag is larger. The variance $\hat{\sigma}_i^2$ is updated according to

$$\hat{\sigma}_{i+1}^2 = \hat{\sigma}_i^2 - K \cdot \hat{\sigma}_i^2 \tag{6.37}$$

  Note that such a Kalman filter based method corresponds to a Gaussian approximation of the posterior distribution $P(\mathbf{x}_{t+1}|\mathbf{y}_{t+1})$ in (6.21).

- *Configuration 3.* The second method is used to localize the agent with the two closest visible flags.
- *Configuration 4.* The second method is applied to localize the agent using all possible combinations of two visible flags. The resulting estimates are again weighted depending on the distances to the perceived flags and updated according to the Kalman filter update equation (see (6.35)). However, since each measurement is generated using information about two visible flags, we get two ranges of possible distance values on which to base the weight. We have already seen that the perceived distance d' to a flag defines a circle of possible agent positions. As a result, the range $[r_{min}, r_{max}]$ of real distances corresponding to r' can be seen to define a ring-like area that contains the agent's true position. Combining the information about two flags thus gives us an intersection area of two rings in which the agent's position must be located. Clearly, the size of the intersection area depends on the possible distance ranges and represents a measure for the confidence in the estimate (as the area gets larger, the weight should decrease since the measurement is taken from a larger set of possibilities). The size of this intersection area is therefore used to weight the samples (i.e. it represents σ_{i+1}^2 in (6.36)). This resembles a spherical Gaussian that is centered on the mean of this area.
- *Configuration 5.* The particle filter method is applied to determine the agent's global position. We use the second method (fourth configuration) to obtain an initial position estimate[5] and center a grid of 13×13 particles on this estimate with $0.0216m$ between them (this distance will be explained later). After each update, the original number of particles is restored by resampling from the posterior using the Gaussian sampling method described earlier. When all the visible flags have been processed, the remaining set of particles is averaged to obtain our estimate of the agent's global position.
- *Configuration 6.* The agent's global position is again determined by applying the particle filter algorithm and using the second method (fourth configuration) to obtain an initial estimate. The difference with the previous configuration is that the resampling step is now omitted, whereas the number of particles is increased in such a way that the total computation time for both configurations (5 and 6) is the same. It turns out that this is approximately the case for a grid of 21×21 particles (separated by $0.013m$ to keep a constant size). The idea is to investigate whether a more accurate estimate can be obtained in the same amount of time by using more points which are not resampled.

These configurations were tested by placing the agent 10,000 times at a random position on the field with a random orientation. After each placement, the agent calculated his global position based on the visual information that he received. This was done for each configuration. The resulting estimates were then compared to the noise-free global information received by the coach to determine the estimation error for each configuration on that particular trial. Table 6.2 displays the results of this experiment which was performed on a Pentium III 1GHz/256MB machine running Red Hat Linux 7.1.

Configuration	Average error (m)	Standard deviation (m)	Total time (s)
1	0.1347	0.1270	0.0542
2	0.1485	0.2026	0.4542
3	0.2830	0.5440	0.0373
4	0.0821	0.0516	15.0893
5	0.0570	0.0442	56.6959
6	0.0565	0.0440	54.2441

Table 6.2: Localization performance for different configurations over 10,000 iterations.

[5]The setup for the localization experiment (random placement) was such that previous agent positions could not be used.

These results clearly show that configurations 5 and 6 outperform the alternatives with respect to the accuracy of the estimates. The average error for the first configuration (using the furthest visible line and closest visible flag) is more than twice as high although the method is faster. When the first method is used with all the visible flags (configuration 2) the total time obviously increases and the average error becomes larger too. The reason for this is that the estimate is now partly based on visual information about flags located at a larger distance from the agent than for the first configuration. More noise is thus incorporated into the information that is used and despite the fact that these samples are weighted less in the overall estimate they have a negative influence on the estimation error. The average error for the third configuration (using the two closest flags) is the highest. In general, the second method is faster than the first but less accurate. This can be explained due to the fact that the noise included in the visual observations defines a larger area of possible agent positions for the second method than for the first method. The estimate obtained by applying the second method will thus be one from a larger set of possibilities and this increases the possible estimation error. However, if we use all possible combinations of two visible flags and weight the resulting samples based on the distances to these flags (configuration 4) the estimate is improved significantly. When the agent is placed at a random position on the field as was done in the experiment, he sees 13.4 flags on average with a standard deviation of 7.2. This gives a total number of approximately $\binom{13}{2} = 78$ pairs of flags on which to base the estimate[6]. The weighted average is thus taken over a large number of measurements and this causes the noise to be filtered out.

It can be concluded that the fourth configuration outperforms the first three in terms of accuracy. This justifies our choice of using the fourth configuration to obtain an initial position estimate for the particle filter algorithm used in configurations 5 and 6. Note that the total width of the particle grid ($0.26m$) has been chosen approximately equal to twice the sum of the average error and standard deviation for the fourth configuration. In this way, the particle grid will almost certainly cover the agent's true position when centered on the initial estimate. Configurations 5 and 6 clearly yield the best results: they achieve a very low estimation error[7] with a small standard deviation. This is not surprising however, since the filter uses visual information about all the flags to further improve the initial estimate which is already very good. Note that this obviously makes the method significantly slower than the previous ones. However, when we consider the fact that the total time applies to 10,000 iterations the method is still more than fast enough to meet the agent's real-time demands. Note that there is hardly any difference between the fifth configuration, which resamples the particles, and the sixth configuration, which does not resample but uses more points. Both configurations achieve about the same error in equal time. Due to the slightly better estimation error for the sixth configuration, the *UvA Trilearn* agents use this configuration to localize themselves. It turns out that our agent localization method is very accurate compared to those of other teams which report estimation errors in the range $0.12 - 0.3$ ([16, 27, 91, 106, 111]).

6.4.2.2 Agent Orientation

The agent can also use the visible landmarks on the field to determine his global neck angle and global body direction. Here we present two methods which can be used to calculate the agent's global neck angle.

- The first method uses the perceived direction to the furthest visible line to determine the agent's global neck angle. This method has been described in Section 6.4.2.1 during the explanation of the first agent localization method. Note that the resulting angle includes noise due to the fact that the actual direction to the line is quantized by rounding the value to the nearest integer.

[6]During matches this number will be even larger since the agent is then often positioned more centrally on the field which enables him to see more flags. However, the random placement sometimes causes agent positions from which only a small number of flags are visible.

[7]Especially when one considers the fact that the field has a size of $105m \times 68m$.

- The second method uses the perceived direction $\tilde{f}_\phi$ to a flag f and combines this information with the known global position (f_x, f_y) of f and with the calculated global position (p_x, p_y) of the agent to determine the agent's global neck angle. The first step is to compute the angle α between (f_x, f_y) and (p_x, p_y). The agent's global neck angle θ is then obtained by subtracting from α the relative direction $\tilde{f}_\phi$ to the flag. In summary, the global neck angle θ can thus be calculated as follows:

$$\theta = normalize(\text{atan2}(f_y - p_y, f_x - p_x) - \tilde{f}_\phi) \tag{6.38}$$

  where *normalize* is a function that converts the resulting angle to an equivalent angle from the interval $[-180, 180]$ and where 'atan2(x,y)' is a function that computes the value of the arc tangent of x/y in the range $[-180, 180]$ using the signs of both arguments to determine the quadrant of the return value. Note that the resulting estimate again includes noise since the directional observation $\tilde{f}_\phi$ is the result of rounding the real value to the nearest integer. Furthermore, the quality of the estimate will also depend on the accuracy of the calculated global position (p_x, p_y). The results presented in the remainder of this section are based on estimates of (p_x, p_y) obtained from using the sixth configuration described in Section 6.4.2.1.

We have compared the performances of the two methods described above by performing an agent orientation experiment. For this experiment we used three different configurations:

- *Configuration 1.* The first method is applied to determine the agent's global neck angle using the perceived direction to the furthest visible line.
- *Configuration 2.* The second method is applied to determine the agent's global neck angle using the perceived direction to the closest visible flag.
- *Configuration 3.* The second method is applied to determine the agent's global neck angle using all the visible flags. For each flag the perceived direction is combined with the flag's known global position and with the global position of the agent to obtain an estimate of the agent's global neck angle. The overall estimate is then computed by averaging the measurements for all the flags. Note that it is not necessary to weight the samples in this case since noise is added to each observation in the same way (i.e. by rounding the real direction to the flag to the nearest integer).

These configurations were again tested by placing the agent 10,000 times at a random position on the field with a random orientation. After each placement he then calculated his global neck angle using the visual information that he received. This was done for each configuration. The resulting estimates were then compared to the noise-free global information received by the coach to compute the estimation error on each trial[8]. Table 6.3 displays the results of this experiment which was performed on a Pentium III 1GHz/256MB machine running Red Hat Linux 7.1. These results show that the first method is more accurate and slightly faster than the second (compare configurations 1 and 2). However, when the second method is used with all the visible flags (configuration 3) the estimation error is significantly reduced while the increase in total computation time over 10,000 iterations is negligible. Note that this scaling approach is not possible for the first method, since the agent always sees only a single line when he is positioned inside the field. The *UvA Trilearn* agents use the third configuration to determine their global neck angle. They can then calculate their global orientation (i.e. body angle) by combining this neck angle with the $\tilde{\theta}^t$ information which is contained in **sense_body** messages and which denotes the agent's neck angle relative to his body. Note that in this case it would also have been possible to improve the neck angle estimate through particle filtering. We chose not to do this however, since the average error for the third configuration was already extremely small[9].

[8] Actually, the coach also receives directions as rounded integers. For this experiment we therefore had to adapt the *soccer server* implementation in such a way that the real directional values were sent to the coach to enable the comparison.

[9] For comparison: *YowAI*, a strong simulation team from recent years, reports angle estimation errors of ± 0.5 degrees [106].

Configuration	Average error (deg)	Standard deviation (deg)	Total time (s)
1	0.5038	0.2894	0.0115
2	0.5569	0.4206	0.0212
3	0.1898	0.1746	0.1990

Table 6.3: Neck angle estimation performance for different configurations over 10,000 iterations.

6.4.2.3 Dynamic Object Information

When the agent's global position and orientation have been updated, this information is used to update the world model attributes for other dynamic objects (i.e. the ball and other players). For these objects the following values need to be computed:

- The global position (q_x, q_y).
- The global velocity (v_x, v_y).
- The global body angle θ^t_{body} (only for players)
- The global neck angle θ^t_{neck} (only for players)

The global position (q_x, q_y) of a dynamic object can be determined by combining the relative visual information about the object with the derived global information about the agent. The visual message effectively contains the relative position $(\tilde{q}_r, \tilde{q}_\phi)$ of the object in polar coordinates. In order to determine the object's global position we must align the relative coordinate system (polar) with the global coordinate system (Cartesian) by performing a rotation. Since all visual information is relative to the agent's neck, the rotation angle equals the agent's global neck angle θ which has been computed earlier. The polar coordinates must then be converted to Cartesian coordinates and the result must be added to the agent's global position (p_x, p_y). This gives the following formula for calculating the object's global position (q_x, q_y):

$$(q_x, q_y) = (p_x, p_y) + \pi(\tilde{q}_r, \tilde{q}_\phi + \theta) \tag{6.39}$$

where π is a function that converts polar coordinates to Cartesian coordinates as shown in Equation 6.14. Note that we do not use past observations to improve the estimate of the object's global position. The reason for this is that the agent has no information about the actions that have been performed by other players. There is thus no way to determine the relevance of old positional information about dynamic objects (players might have dashed since then, the ball might have been kicked, etc.). As a result, the current visual perception is the only completely reliable source of information about the position of other dynamic objects on the field.

If the distance to a visible player is not too large, a **see** message also contains the body direction $\tilde{\theta}^t_{body}$ and neck direction $\tilde{\theta}^t_{neck}$ of this player relative to the neck direction of the agent. These values can be used to calculate the player's global body angle θ^t_{body} and global neck angle θ^t_{neck} by simply adding the agent's global neck angle θ to the relative directions included in the visual message:

$$\theta^t_{body} = \tilde{\theta}^t_{body} + \theta \tag{6.40}$$

$$\theta^t_{neck} = \tilde{\theta}^t_{neck} + \theta \tag{6.41}$$

Note that if the visual message does not contain $\tilde{\theta}^t_{body}$ and $\tilde{\theta}^t_{neck}$ (i.e. the player is too far away) no assumptions are made concerning the player's behavior in the previous cycle. This means that the values for θ^t_{body} and θ^t_{neck} which were already stored in the world model remain unchanged.

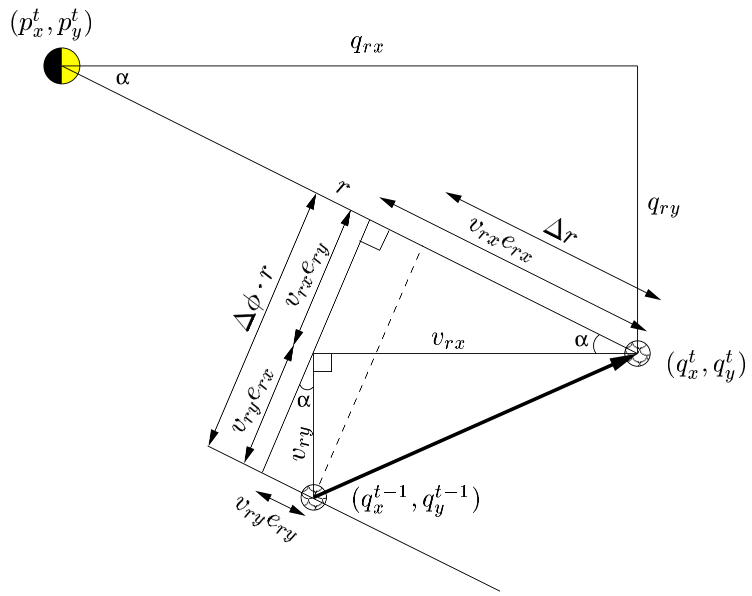


Figure 6.6: Components used for calculating change information of dynamic objects.

Estimating the velocity of dynamic objects is a prominent part of updating the agent's world model. An accurate estimate of the ball velocity, for example, is very important for determining the optimal interception point. We describe three methods for estimating the velocity of dynamic objects which will be compared using different configurations. In the remainder of this section we will concentrate on estimating the ball velocity, although the methods presented can be applied to players as well.

Method 1: velocity estimation using change information

The first method uses the distance change Δr and direction change $\Delta\phi$ which are included in a visual message to estimate the ball velocity (v_x, v_y). Recall from Section 3.2.1 that the values for Δr and $\Delta\phi$ are calculated according to the following equations:

$$\Delta r = (v_{rx} \cdot e_{rx}) + (v_{ry} \cdot e_{ry}) \tag{6.42}$$

$$\Delta\phi = [(-(v_{rx} \cdot e_{ry}) + (v_{ry} \cdot e_{rx}))/r] \cdot (180/\pi) \tag{6.43}$$

where r and (v_{rx}, v_{ry}) respectively denote the distance to the ball and the ball velocity relative to the agent's neck direction. Furthermore, (e_{rx}, e_{ry}) is equal to the unit vector in the direction of the relative position (q_{rx}, q_{ry}) of the ball:

$$e_{rx} = q_{rx}/r \tag{6.44}$$

$$e_{ry} = q_{ry}/r \tag{6.45}$$

where $(q_{rx}, q_{ry}) = (q_x, q_y) - (p_x, p_y)$, i.e. the relative ball position equals the difference between the ball's global position (q_x, q_y) and the agent's global position (p_x, p_y). A graphical depiction of these quantities is shown in Figure 6.6 for an example situation. The distance change Δr can be regarded as the difference between the x-coordinates for the current and previous ball positions in the relative coordinate system of the agent, whereas the direction change $\Delta\phi$ can be regarded as the quotient of the difference between the y-coordinates of these positions and the distance r to the ball. Given the distance change Δr, direction change $\Delta\phi$ and distance r, it is possible to recover the relative ball velocity (v_{rx}, v_{ry}) by substituting (6.42) into (6.43) and rewriting the result. This gives:

$$v_{rx} = \Delta r \cdot e_{rx} - \Delta\phi \cdot (\pi/180) \cdot r \cdot e_{ry} \quad (6.46)$$
$$v_{ry} = \Delta r \cdot e_{ry} + \Delta\phi \cdot (\pi/180) \cdot r \cdot e_{rx} \quad (6.47)$$

Since all visual information is relative to the agent's global neck angle θ, the global ball velocity (v_x, v_y) can now be obtained by rotating the relative velocity vector (v_{rx}, v_{ry}) over θ degrees.

Note that the resulting velocity estimate (v_x, v_y) is based on the assumption that the environment is noiseless, whereas the server in fact adds noise to the simulation in several ways. Two forms of noise actually affect the accuracy of the ball velocity estimate. These are:

- Movement noise. In order to reflect unexpected movements of objects in the real world, the *soccer server* adds uniformly distributed random noise to the movement of the ball in each cycle as was shown in Section 3.3. The amount of noise that is added depends on the speed of the ball, i.e. when the ball goes faster more noise will be added. It is important to note that movement noise actually changes the direction of the velocity vector and is not just included into sensory observations.
- Sensor noise. One of the real-world complexities contained in the *soccer server* is that noise is incorporated into the visual sensor data by quantizing the values sent by the server. Examples are that the precision of visual information about an object decreases as the distance to this object increases and that angles are rounded to the nearest integer. The values for the distance, direction, distance change and direction change are quantized according to Equations 3.15−3.19. Given the quantized values it is possible to determine the ranges from which the real values must originate by inverting these equations accordingly. This range information can then be used for recursive parameter estimation methods or filtering algorithms to improve the velocity estimate.

Method 2: position-based velocity estimation

The second method uses the difference between the global position (q_x^{t-1}, q_y^{t-1}) of the ball in the previous cycle and its current global position (q_x^t, q_y^t) to estimate the global ball velocity (v_x^t, v_y^t) in the current cycle. Since the agent has no information concerning the actions performed by other players, it is assumed that the ball has not been kicked by any of them. Equation 6.2 shows that when the ball has not been kicked (i.e. has zero acceleration) the movement (u_x^t, u_y^t) from cycle $t-1$ to cycle t is equal to the velocity (v_x^{t-1}, v_y^{t-1}) in cycle $t-1$. From this it can be concluded that (v_x^{t-1}, v_y^{t-1}) equals the difference between (q_x^t, q_y^t) and (q_x^{t-1}, q_y^{t-1}). In order to compute the velocity in cycle t, we have to multiply the velocity in cycle $t-1$ by the velocity decay rate of the ball which is represented by the server parameter `ball_decay`. This gives the following formula for calculating the velocity (v_x^t, v_y^t) based on consecutive ball positions:

$$(v_x^t, v_y^t) = ((q_x^t, q_y^t) - (q_x^{t-1}, q_y^{t-1})) \cdot \texttt{ball\_decay} \quad (6.48)$$

Method 3: velocity estimation using a particle filter

The third method uses the particle filter algorithm described in Section 6.4.2.1 to estimate the current ball velocity. Each particle is represented by a quadruple (x, y, v_x, v_y) which contains the relative position and velocity of the ball in Cartesian coordinates. When the first visual observation is received by the agent, the included distance and direction to the ball are used to determine the ranges $[r_{min}, r_{max}]$ and $[\phi_{min}, \phi_{max}]$ from which the real values must originate. This is done by inverting (3.15) and (3.18) using the 'inverse' quantize range $[V_{min}, V_{max}]$ defined by (6.28) and (6.29). The same is done for the distance change and direction change information which yields the ranges $[\Delta r_{min}, \Delta r_{max}]$ and $[\Delta\phi_{min}, \Delta\phi_{max}]$. These four ranges are used to initialize a set of particles $(r, \phi, \Delta r, \Delta\phi)$ by randomly selecting a value from each range. After this, the polar coordinates r and ϕ are converted to Cartesian coordinates x and y according to Equation 6.14 and the values for Δr and $\Delta\phi$ are used to compute a (v_x, v_y) estimate using the first method described above. This results in a set of particles (x, y, v_x, v_y). The initial velocity estimate is now obtained by computing the average of the velocity components of all the particles.

At the start of the next cycle, the position and velocity information contained in each particle is updated according to the known *soccer server* dynamics and noise model. For this we use the exact same equations that the server uses, meaning that uniformly distributed random noise is added in the same way. When a new visual message arrives, it is then checked for each particle whether *all* the values (i.e. position[10] and velocity) are possible based on the new information. This is done by computing the ranges of possible values for r, ϕ, Δr and $\Delta\phi$ for the current observation and by checking whether all the particle values fall within the corresponding ranges. If for a certain particle this is not the case then this particle is removed from the set. After this, the particle set is resampled by randomly selecting one of the remaining particles and making an exact copy of it[11]. This is repeated until the original number of particles has been restored. The velocity estimate is then equal to the average of the velocity components of all the particles. Note that it is possible that the new observation causes all the particles to be removed from the set (e.g. because the ball has been kicked which completely changes its position and velocity). In this case, the possible value ranges corresponding to the current observation are used to re-initialize the particle set.

Note that this particle filter differs from the one which was used in the agent localization experiment in Section 6.4.2.1. In the method described there all the particles were reinitialized for each subsequent position estimate. In the current case however, they are propagated from each step to the next and only reinitialized when the particle set becomes empty. Another difference is that more particles will be needed for the filter described in this section since the state space has now become 4D instead of 2D.

Results

We have compared the performances of the three methods described above by conducting a velocity estimation experiment. For this experiment we used four different configurations:

- *Configuration 1.* The first method is applied to estimate the ball velocity while neglecting the movement noise of the ball as well as the noise contained in the visual sensor data.
- *Configuration 2.* The first method is applied to estimate the ball velocity thereby taking past estimates into account as well as the noise that is added to visual observations. This is again done according to the Kalman filter principles discussed earlier. However, the main difference in this case is that the velocity estimate is now updated in two different ways. The first update takes place at the start of a new simulation cycle (i.e. when the agent receives a **sense_body** message). Although new visual information has not arrived at this stage it is still possible to update the velocity estimate according to the known *soccer server* dynamics. This means that the estimated ball velocity $\hat{\mathbf{v}}^{t-1}$ in the previous cycle is multiplied by the speed decay of the ball to obtain a new estimate as follows:

$$\hat{\mathbf{v}}^t = \hat{\mathbf{v}}^{t-1} \cdot \texttt{ball\_decay} \tag{6.49}$$

[10] Note that we include the position information in a particle, since the movement of the ball from one cycle to the next says something about its velocity as is shown in the second method. Furthermore, this position information might provide a good alternative to Equation 6.39 for estimating the global ball position as well. We are currently working in this direction.

[11] Note that identical particles that result from the resampling step will be spread out again in the next cycle due to the fact that the noise added during the particle update has a random character.

Note that the movement noise which is added by the server is neglected in this equation. However, the added noise in x- and y-direction (which is non-white due to the fact that it depends on the speed of the ball in the previous cycle) is taken into account by increasing the variance $\hat{\sigma}^2_{t-1}$ associated with the previous estimate in the following way (Equation 3.24 clarifies this):

$$\hat{\sigma}^2_t \quad = \quad \hat{\sigma}^2_{t-1} \cdot \texttt{ball\_decay}^2 + \frac{(2 \cdot \texttt{ball\_rand} \cdot \|\hat{\mathbf{v}}^{t-1}\|)^2}{12} \cdot \frac{(2 \cdot \texttt{ball\_rand} \cdot \|\hat{\mathbf{v}}^{t-1}\|)^2}{12} \qquad (6.50)$$

The second update is performed when the agent receives new visual information. To this end, the ranges of possible values for the distance change and direction change are calculated from the quantized values received in the visual message. This gives possible ranges $[\Delta r_{min}, \Delta r_{max}]$ and $[\Delta\phi_{min}, \Delta\phi_{max}]$ from which the perceived values must originate. The first method is now applied using the mean of these two ranges (for which the values are uniformly distributed) to compute a prediction $\mathbf{v}^t$ of the current ball velocity. This prediction is then weighted with the current estimate $\hat{\mathbf{v}}^t$ to obtain an improved estimate of the velocity of the ball in the current cycle. This gives:

$$\hat{\mathbf{v}}^t \quad := \quad \hat{\mathbf{v}}^t + K \cdot (\mathbf{v}^t - \hat{\mathbf{v}}^t) \qquad (6.51)$$

where the correction term K is again equal to

$$K \quad = \quad \frac{\hat{\sigma}^2_t}{\hat{\sigma}^2_t + \sigma^2_t} \qquad (6.52)$$

Here the variance $\hat{\sigma}^2_t$ is updated according to (6.37) and the variance σ^2_t of the current measurement is determined using the ranges $[\Delta r_{min}, \Delta r_{max}]$ and $[r \cdot \Delta\phi_{min}, r \cdot \Delta\phi_{max}]$ (see Figure 6.6) of possible ball coordinate changes in the relative coordinate frame of the agent. Since the values in these ranges are uniformly distributed this gives:

$$\sigma^2_t = \frac{(\Delta r_{max} - \Delta r_{min})^2}{12} \cdot \frac{(\Delta\phi_{max} - \Delta\phi_{min})^2 \cdot r^2}{12} \qquad (6.53)$$

- *Configuration 3.* The second method is applied to estimate the ball velocity using information about the global position of the ball in consecutive cycles.
- *Configuration 4.* The third method (particle filter) is applied to estimate the ball velocity using a particle set that consists of 300 particles.

These configurations were tested by means of the following experiment which is depicted in Figure 6.7. Two players were placed on the field respectively at positions $(-14, 0)$ and $(0, 14)$ both facing towards the center spot. During each iteration the second player shot the ball with random power in a random

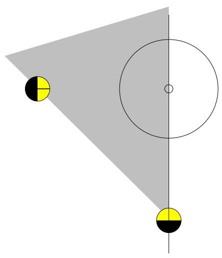


Figure 6.7: The setup for the velocity estimation experiment.

Configuration	Average error (%)	Standard deviation (%)
1	8.8536	6.8835
2	6.7963	7.116
3	48.360	682.1773
4	5.9075	5.3537

Table 6.4: Velocity estimation results for different methods over 1,500 iterations (18,121 measurements).

direction between 0 and −45 degrees (the shaded area in Figure 6.7 shows the part of the field to which the ball is shot). The first player did not move and used the visual information that he received to estimate the ball velocity in each subsequent cycle until the ball stopped or moved outside his view cone. This was done for each of the four configurations described above. The resulting estimates were then compared to the noise-free global information received by the coach to determine the relative estimation error for each configuration. This experiment was repeated 1,500 times leading to a total of 18,121 measurements. Note that the positions of both players were chosen in such a way that the distance between them was less than the value for the server parameter `unum_far_length` (currently 20). This meant that the first player would always receive the distance change and direction change information about the ball in a visual message (see Section 3.2.1). Furthermore, the players were put at the edges of each other's view cones to ensure that the first player would be able to see the ball for as long as possible.

Table 6.4 displays the results of this experiment which was performed on a Pentium III 1GHz/256MB machine running Red Hat Linux 7.1. These results clearly show that the position-based velocity estimation method (configuration 3) yields the largest error. This is caused by the sensor noise that is added to the true distance and direction values as observed by the player. This noise increases significantly as the distance to the ball increases which leads to a large error in the estimated ball position. The results become much better when the distance change and direction change information included in visual messages is used to estimate the ball velocity (first method). Even when the noise is completely neglected and past observations are ignored (configuration 1), this method performs much better than the position-based algorithm. The results can be further improved by computing the velocity as a combination of an estimate obtained from the current perception (taking noise into account) and a prediction based on past observations (configuration 2). In this case the error is reduced to about 6.8%. However, the particle filter method (configuration 4) yields the best results. This can be explained due to the fact that this method takes both the visual sensor noise and ball movement noise into account. The calculation of possible value ranges from which the values included in visual messages must originate provides an effective way of initializing and filtering the particle set. Furthermore, the possible movement noise is incorporated by simulating the *soccer server* dynamics including the noise model during each particle update. Each particle can thus be seen as a possible ball state and when the number of particles is chosen large enough the average of these possible states represents an accurate estimate of the true situation.

Figure 6.8 shows the results of the velocity estimation experiment as a function of the distance for the first, second and fourth configurations. It is clear from these plots that the average estimation error decreases as the distance to the ball becomes smaller. This is caused by the fact that the noise in the perceived values for the distance and distance change is larger when the ball is further away. Furthermore, the setup for the velocity estimation experiment was such that the ball always moved towards the player that calculated the velocity. During a single iteration the distance from this player to the ball therefore decreased over time. As a result, estimation methods which used past observations to estimate the current ball velocity (configurations 2 and 4) achieved better results as the distance got smaller since the estimates from the previous cycles then became more accurate. This effect is clearly visualized in Figure 6.8: shortly after the ball has been kicked by the second player, the average error and standard deviation

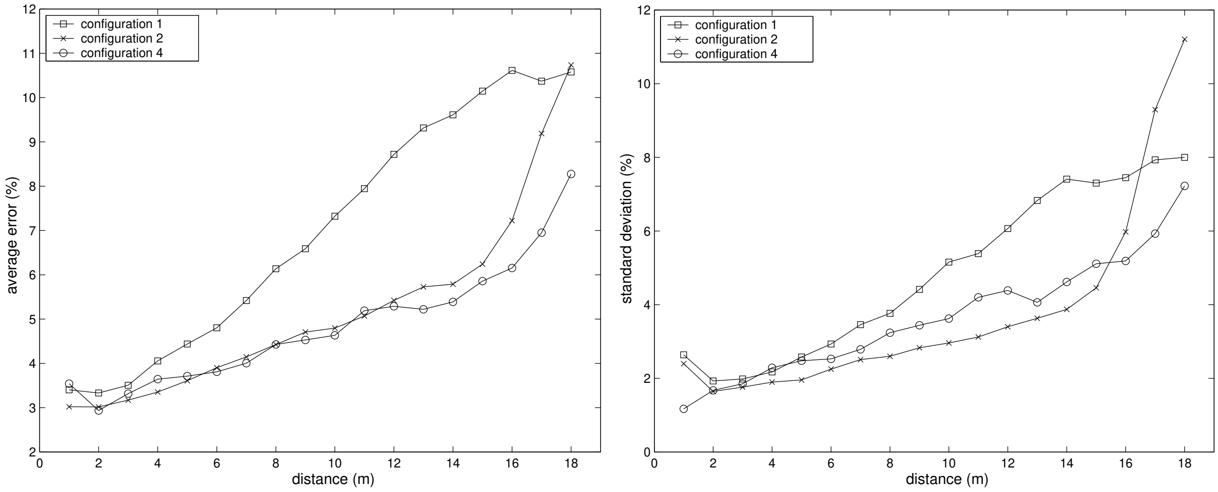


Figure 6.8: Average velocity estimation error and standard deviation as a function of the distance.

for the second configuration drop rapidly as the distance decreases. This is also the case for the fourth configuration although not quite as significantly. Note that it can be concluded from these results that for small distances (< 12) the results for the second configuration are about as good as those for the particle filter algorithm. Nevertheless, the *UvA Trilearn* agents use the particle filter algorithm for estimating the velocity of dynamic objects due to the better estimation error for larger distances.

6.4.3 Update from Aural Sensor

The aural sensor detects spoken messages which are sent when a player or coach issues a **say** command. Calls from the referee are also treated as aural messages. Aural messages contain the following information:

- The time t (i.e. the current cycle) at which the information applies.
- The sender of the message (**referee**, **online_coach_left**, **online_coach_right** or **self**) or the relative direction to the sender if the sender is another player. Note that no information is given about which player sent the message or about the distance to the sender.
- A string with a maximum length of 512 bytes representing the contents of the message.

Referee messages mostly contain information concerning changes in the current play mode. This happens, for example, when a goal has been scored or when the ball goes out of bounds. As soon as such a message arrives it is processed by storing the given information at the appropriate location in the agent's world model. No actions then have to be taken during the *update* phase. Messages from the coach usually contain advice for the players based on observations about the current match. The players can use these messages to adapt their strategy. Currently, the *UvA Trilearn* coach is only used for selecting heterogeneous player types for certain field positions and for changing the players at these positions. The players are informed of these changes by the referee. The coach himself thus passes no messages to the players.

The *soccer server* communication paradigm models a crowded, low-bandwidth environment in which the agents from both teams use a single, unreliable communication channel. Spoken messages have a limited length (512 characters) and are only heard by players within a certain distance (50 metres) from the speaker[12]. Furthermore, each player can hear only one message from a teammate every two simulation cycles. Note that the permitted message length of 512 characters makes it possible to communicate a considerable amount of information. It enables the agents, for example, to communicate their world model to other teammates which would clearly not be possible if only short messages were allowed. The *UvA Trilearn* agents therefore use communication to gather information about the parts of the field which are currently not visible to them. The agent which has the best view of the field in a given situation communicates his world model to other team members which can then use this information to increase the reliability of their own world state representation. In the remainder of this section we will describe the exact syntax of these messages and the way in which they are processed when received by an agent. The policy that determines which player should communicate in a given situation and that defines how frequently the broadcasting should take place will be discussed in Chapter 9.

When a player communicates world state information to other teammates, the message first of all contains the side (left or right team), current cycle and uniform number of the speaker. Since all 22 players on the field use the same communication channel, this part of the message is encoded in such a way that it can be recognized that the message comes from a teammate and not from an opponent. Furthermore, this encoding scheme also protects the agents from communicative interference by opponent players. Opponents could, for example, try to corrupt the world model of an agent by re-sending previously spoken messages from the other team at a later time. However, the coding scheme prevents damage in these situations since the encoded cycle number will reveal that the information in the message is outdated. The encoded part of the message is followed by the global position and velocity of the speaker and of the ball for which the values are accompanied by their corresponding confidence values. The remainder of the message then contains positional information about other players for which the confidence is higher than a certain threshold. For each player the information consists of his uniform number followed by his global position and associated confidence value. In order to indicate that a player is a goalkeeper (which is important to know), an optional '`g`' is added after his uniform number. This is necessary since goalkeepers do not always have the same uniform number. If the uniform number of a player is unknown due to the fact that the distance to this player is too large, then the uniform number for the player is set at -1. Note that information about objects for which the associated confidence is low is not communicated.

The grammar for the message syntax is shown in Table 6.5. This shows that the encoded time stamp in the message takes the form '`[a-j][c-l][e-n][g-p]`'. The encoding is such that a letter is chosen from each of these four ranges based on the digits of the current cycle number. Each range consists of 10 consecutive letters which respectively correspond to the numbers 0 to 9. For example, cycle number '0000' will be encoded as '`aceg`', whereas cycle number '1934' will become '`blhk`'. The encoding scheme for the uniform number of the speaker is analogous: a letter is chosen from the range `[j-t]` in which the consecutive letters correspond to the numbers 1 to 11. Note that double values that represent position and velocity coordinates are only specified up to one digit behind the decimal point in order to make sure that the message string will not exceed the maximum length of 512 bytes. When an agent receives aural information, it is first determined whether the message was sent by a teammate by looking at the first letter which denotes the side of the sender. If the side matches that of the agent, it is checked whether the message contains up-to-date information by decoding the encoded time stamp. If it turns out that the time stamp corresponds to the current cycle, the world state information is extracted from the message and compared to the information that is currently stored in the agent's world model. Object information from the message is then only inserted into the model if the associated confidence value is higher than the confidence in the current information for that object. If positional information is included about a player for which the uniform number is unknown, the agent looks for the closest player based on the current world state information. If the distance between this closest player and the anonymous player

[12] Referee messages and messages from the coach are an exception since these can be heard by all the players.

```
<MESSAGE>        -> <ENCODING> <AGENT_INFO> (<BALL_INFO>) (<TEAM_INFO>) (<OPP_INFO>)

<ENCODING>       -> <SIDE> <ENC_TIME> <ENC_UNIFORM_NR>
<SIDE>           -> l | r
<ENC_TIME>       -> [a-j][c-l][e-n][g-p]
<ENC_UNIFORM_NR> -> [j-t]

<AGENT_INFO>     -> <GLOB_POS> <GLOB_VEL>
<GLOB_POS>       -> [double] [double]
<GLOB_VEL>       -> [double] [double]

<BALL_INFO>      -> <GLOB_POS> <GLOB_VEL> <CONF>
<CONF>           -> double(0..1)

<TEAM_INFO>      -> <UNIFORM_NR> <IS_GOALIE> <GLOB_POS> <CONF> <TEAM_INFO> | null
<UNIFORM_NR>     -> [1-11] | -1
<IS_GOALIE>      -> g | null

<OPP_INFO>       -> <UNIFORM_NR> <IS_GOALIE> <GLOB_POS> <CONF> <OPP_INFO> | null
```

Table 6.5: Grammar for the *UvA Trilearn 2001* message syntax for inter-agent communication.

position that is contained in the message is smaller than a certain threshold, the new position is assigned to this closest player. If the distance exceeds this value however, the information is assigned to the player with the lowest confidence below another threshold. Note that the entire message is discarded if the side indicator at the start of the message does not match that of the receiver or if the time stamp is outdated (i.e. is not equal to the current cycle time). Also note that in case of an aural message no calculations have to be performed during the *update* phase.

6.5 Prediction Methods

Prediction methods can be used to predict future states of the world based on past perceptions. These methods are important for the action selection process. The agent world model currently contains only low-level prediction methods, i.e. methods for predicting low-level world state information such as future positions of dynamic objects. Predictions at a more abstract and higher level (e.g. about the success of a certain strategy or the behaviour of opponent players) cannot be made however. The prediction methods that are contained in the world model can be divided into three main categories:

- Methods for predicting the state of the agent after performing a certain action. These methods receive an action command as their only input argument and return a prediction of the agent's state (i.e. global position, global velocity, global body angle, global neck angle and stamina) after the given command has been performed. This future state information is calculated by simulating the *soccer server* movement model and action models based on the command parameters. For this we use the server equations presented in Sections 3.3 and 3.4. The noise in these equations is ignored.
- Methods for predicting future states of other dynamic objects on the field. Note that this is very difficult for players due to the fact that the agent has no knowledge about which actions other

players intend to perform. In general, predictions about other players are therefore never made and the methods in this category mainly focus on the ball. Future ball positions (and velocities) are easier to predict since ball movement is less random than player movement. Players can perform many different kinds of actions which influence their state in different ways, whereas the ball generally moves along a certain trajectory which can only be changed by kicking it. It is possible to estimate the ball positions in consecutive cycles by making use of the known *soccer server* dynamics for updating the state of the ball. A key observation in this respect is that ball movement can be modeled by a geometric series. A geometric series is defined as a series in which the ratio between consecutive terms is constant. This can be expressed as follows:

$$a,\ ar,\ ar^2,\ ar^3,\ \ldots \tag{6.54}$$

where a will be referred to as the first term and r as the ratio of the series. We have already seen that the *soccer server* simulates the movement of the ball from one cycle to the next by adding the ball velocity to its current position and by decreasing the velocity at a certain rate. Since the ball velocity decays at the same rate in each cycle, it is possible to describe this movement model as a geometric series in which the consecutive terms represent successive ball velocities during the movement. The ratio r of the series is then equal to the value of the server parameter `ball_decay` which represents the velocity decay rate of the ball in each cycle. Clearly, this model can also be used to predict the movement of the ball a number of cycles into the future. If a equals the ball velocity in the current cycle then the velocity after n cycles will be equal to ar^n where $r =$ `ball_decay`. Since the ball movement from one cycle to the next is obtained by adding the velocity vector to the current ball position, we can also determine the distance that the ball has traveled during this period by adding up the consecutive ball velocities. To this end we need to compute the sum of the geometric series. Assuming that we want to know the distance that the ball has traveled over a period of n cycles, this sum can be calculated as follows:

$$\sum_{i=0}^{n} ar^i \quad = \quad a \cdot \frac{1-r^n}{1-r} \tag{6.55}$$

This gives us a simple closed formula for computing the traveled distance. Furthermore, it enables us to predict future positions of the ball after an arbitrary number of cycles by adding the result of (6.55) to the current ball position. This leads to the following formula for predicting the global ball position (p_x^{t+n}, p_y^{t+n}) after n cycles:

$$(p_x^{t+n}, p_y^{t+n}) \quad = \quad (p_x^t, p_y^t) + \pi(v_r^t \cdot \frac{1 - \texttt{ball\_decay}^n}{1 - \texttt{ball\_decay}}, v_\phi^t) \tag{6.56}$$

where (v_r^t, v_ϕ^t) denotes the ball velocity in cycle t in polar coordinates and where π is a function that converts polar coordinates to Cartesian coordinates as shown in Equation 6.14. Note that for the prediction we neglect the noise that is added to the movement of the ball since we cannot use observations from the future to 'filter' this noise. The best prediction therefore lies at the mean of the uniform noise distribution. Furthermore, we also neglect the fact that the ball might be kicked by another player within n cycles since we have no knowledge about the intentions of other players.

- A method for predicting how long it will take a player to reach a certain position on the field. This method receives a target position and a player as its only arguments and returns an estimate of the number of cycles that the player will need to get from his current position to the target position. The estimate is based on the fact that the maximum distance that a player can cover in one cycle (neglecting noise and wind) equals the maximum player speed denoted by the server parameter `player_speed_max`. Note that in general the player will not be able to cover this distance in every cycle, since he needs to turn in the right direction (sometimes more than once due to the noise added to his motion) and cannot reach maximum speed with a single **dash**. A prediction for the number of cycles n that a player needs to get from a position $\vec{p}$ to a position $\vec{q}$ is calculated as follows:

$$n = \mathrm{rint}\left(\frac{\|\vec{q}-\vec{p}\|}{\texttt{player\_speed\_max}} + \frac{|\theta - (\vec{q}-\vec{p})_\phi|}{\mathit{TurnCorrection}}\right) \tag{6.57}$$

where 'rint' is a function that rounds a value to the nearest integer, θ denotes the player's global body angle, $(\vec{q}-\vec{p})_\phi$ corresponds to the global angle between $\vec{p}$ and $\vec{q}$ and *TurnCorrection* is a parameter that represents a correction value for the number of cycles that the player needs to turn in the right direction. Note that the resulting estimate is only intended as a rough approximation.

6.6 High-Level Methods

The agent world model also contains several high-level methods for deriving high-level conclusions from basic world state information. These methods are used to gain more abstract knowledge about the state of the world and to hide some of the *soccer server* details which influence the parameter values of action commands. In this section we will describe the most important high-level methods that are contained in the world model. These can be divided into five main categories:

- Methods that return information about the number of players in a certain area. These methods receive a specific area description as their input and return a value denoting the number of teammates and/or opponents that are located within the given area. This area can be either a circle, rectangle or cone. Examples of such methods are `getNrTeammatesInCircle` and `getNrOpponentsInCone`.
- Methods that return the closest or fastest player to a certain position or object. These methods receive a position or object on the field as input and return the closest or fastest player (either teammate or opponent) to the given input argument. Examples are `getClosestTeammateTo(x,y)` and `getFastestOpponentTo(OBJECT_BALL)`. The former can be determined by iterating over all the teammates and by comparing their global positions to the given position[13]. The latter is computed by iterating over future ball positions in steps of one cycle (using a geometric series as discussed in Section 6.5) and by looking whether opponents can reach these positions before the ball does. The first opponent that can reach a future ball position in an equal or smaller number of cycles than the ball itself is considered to be the fastest opponent to the ball.
- Boolean methods that indicate whether a specific object satisfies a certain high-level constraint. These methods can be used to determine the characteristics of a situation based on which the best possible action can be chosen. The most important examples are:
  - `isBallKickable`: returns *true* if the ball is located within the agent's kickable range.
  - `isBallCatchable`: returns *true* if the ball is located within the goalkeeper's catchable range.
  - `isBallInOurPossesion`: returns *true* if the fastest player to the ball is a teammate.
  - `isBallHeadingToGoal`: returns *true* if the ball is heading towards the goal and has enough speed to reach it (this is relevant knowledge for the goalkeeper).
  - Several methods that return *true* when the current play mode satisfies a certain constraint. Examples are: `isFreeKickThem`, `isGoalKickUs`, `isKickInThem`, etc.

[13] This is actually only done for those teammates for which the associated confidence is higher than a certain threshold

- Methods that return information about angles between players in a certain area. Currently, this category contains only a single method: `getDirectionOfWidestAngleBetweenOpponents(a1,a2,d)`. This method returns the global direction a from the interval $[a1, a2]$ of the safest trajectory for the ball between opponents that are located within a distance d from the current ball position. To this end, the widest angle between opponents in the given area is determined after which the global direction of the bisector of this angle is returned.

- Methods for computing the actual argument that should be supplied to an action command in order to achieve the desired result. These methods use the known *soccer server* equations for manipulating command parameters (see Section 3.4) and invert these equations to determine the parameter value which produces the wanted effect. The most important examples are:

  – `getAngleForTurn(a)`. This method is needed because the actual angle by which a player turns is not equal to the argument supplied to the **turn** command but depends on the speed of the player. As the player moves faster it gets more difficult for him to turn due to his inertia. This method receives the desired turn angle a as input and returns the angle value that must be supplied to the **turn** command to achieve this. This value is computed according to the following formula which is a direct result of inverting Equation 3.31 (neglecting noise):

  $$\text{turn\_parameter} = a \cdot (1.0 + \texttt{inertia\_moment} \cdot \|(v_x, v_y)\|) \tag{6.58}$$

  with `inertia_moment` a server parameter denoting the player's inertia and $\|(v_x, v_y)\|$ equal to the speed (scalar) of the player. Note that if the resulting turn parameter is larger than the maximum turn angle (i.e. > 180 or < -180) then the argument supplied to the **turn** command will be equal to this maximum angle (i.e. 180 or -180).

  – `getActualKickPowerRate()`. This method is needed because the actual power with which a player kicks the ball is not equal to the *Power* argument supplied to the **kick** command but depends on the position of the ball relative to the player. For example, if the ball is located to the player's side and at a short distance away from him, the kick will be less effective than if the ball is straight in front of him and close. This method uses the distance d between the ball and the player and the absolute angle θ between the ball and the player's body direction to determine the actual kick power rate in the given situation. This value is computed according to the following formula (based on (3.26) and (3.27)):

  $$\text{act\_kpr} = \texttt{kick\_power\_rate} \cdot \left(1 - 0.25 \cdot \frac{\theta}{180} - 0.25 \cdot \frac{d}{\texttt{kickable\_margin}}\right) \tag{6.59}$$

  with `kick_power_rate` a server parameter used for determining the size of the acceleration vector and `kickable_margin` a server parameter that defines a player's kickable range.

  – `getKickPowerForSpeed(s)`. This method receives the desired speed s that the agent wants to give to the ball on a kick and returns the kick power that must be supplied to the **kick** command in order to achieve this. This value is computed as follows:

  $$\text{kick\_parameter} = \frac{s}{\text{act\_kpr}} \tag{6.60}$$

  where 'act_kpr' is the actual kick power rate as obtained from (6.59). Note that the kick parameter can become larger than the maximum kick power. In this way, it is possible to identify situations in which the desired speed cannot be reached (see Sections 7.2.8 and 7.3.5).

  – `getKickSpeedToTravel(d,e)`. This method returns the speed that has to be given to the ball on a kick such that it has a remaining speed equal to e after traveling distance d. A geometric series is used to predict the number of cycles n that the ball will need to cover distance d given an end speed of e. Note that we have to reason backwards here, i.e. the first term in the series is denoted by the end speed e and the subsequent terms denote different speeds of the ball before slowing down to e. The ratio r of the series is therefore equal to `1/ball_decay` in this case. We have already seen in Section 6.5 that the traveled distance d amounts to the sum of the consecutive velocities and thus to the sum of the terms in the series. In order to determine the number of cycles n, we need to compute the length of the series given the first term e, ratio r and sum d. This length can be calculated according to the following formula [109]:

$$n = \frac{\log(\frac{d\cdot(r-1)}{e}+1)}{\log(r)} \tag{6.61}$$

The initial speed s that has to be given to the ball can now be calculated as follows:

$$s = \frac{e}{\texttt{ball\_decay}^n} \tag{6.62}$$

- `getPowerForDash(x,y)`. This method receives a *relative* position (x, y) to which the agent wants to move and returns the power that should be supplied to the **dash** command in order to come as close to this position as possible. Since the agent can only move forwards or backwards, the closest point to the target position that the agent can reach by dashing is the orthogonal projection $\vec{p}$ of the target point onto the line l that extends into the direction of his body[14] (forwards and backwards). Let d_1 denote the distance from the agent's current position to $\vec{p}$ and d_2 the maximum distance that a player can cover in one cycle (i.e. d_2 = `player_speed_max`). Then the distance that the agent wants to cover in the next cycle (and thus his desired speed) equals $d = \min(d_1, d_2)$. In order to compute the acceleration that is required to achieve this, we must subtract the agent's current speed in the direction of his body (obtained by projecting the current velocity vector onto l) from d since this distance will already be covered automatically. The parameter for the **dash** command can then be calculated as follows:

$$\text{dash\_parameter} = \frac{d - \pi(v_r, v_\phi - \theta)_x}{\texttt{dash\_power\_rate} \cdot \mathit{Effort}} \tag{6.63}$$

  with (v_r, v_ϕ) the agent's global velocity in polar coordinates, θ the agent's global body angle, π a function that converts polar coordinates to Cartesian coordinates, `dash_power_rate` a server parameter used for determining the size of the acceleration vector and *Effort* the current effort value of the agent representing the efficiency of his actions (see Section 3.4.2). Note that if the resulting dash parameter is larger than the maximum dash power then the argument supplied to the **dash** command will be equal to this maximum power.

[14]This is the x-direction in the agent's relative coordinate frame.

Chapter 7

Player Skills

A skill can be regarded as the ability to execute a certain action. The behavior of an agent depends on the individual skills that this agent can perform. In general, these skills can be divided into simple skills that correspond to basic actions and more advanced skills that use the simple skills as parts of more complex behaviors. The strategy of a team of agents can then be seen as the way in which the individual agent behaviors are coordinated. In this chapter we will describe the skills which are available to the agents of the *UvA Trilearn 2001* robotic soccer simulation team. Together, these skills form a hierarchy consisting of several layers at different levels of abstraction. The skills at the lowest level can be specified in terms of basic action commands which are known to the *soccer server*, whereas the higher-level skills use the functionality offered by the layer below to generate the desired behavior. The execution of each skill is based on the current state of the world which is represented by the agent world model described in Chapter 6. This chapter is organized as follows. In Section 7.1 we give an overview of the different skills layers and present a notation that will be used throughout this chapter to define the various skills. In Section 7.2 we then describe the low-level player skills followed by the intermediate player skills in Section 7.3. Finally, the high-level player skills are presented in Section 7.4. Note that the choice of which skill is selected in a given situation depends on the strategy of the team which will be discussed in Chapter 9.

7.1 Introduction

The skills which are available to the *UvA Trilearn* agents include turning towards a point, kicking the ball to a desired position, dribbling, intercepting the ball, marking opponents, etc. These skills can be divided into different layers which together form a hierarchy of skills. Figure 7.1 shows the *UvA Trilearn* skills hierarchy which consists of three layers. The layers are hierarchical in the sense that the skills in each layer use skills from the layer below to generate the desired behavior. The bottom layer contains low-level player skills which can be directly specified in terms of basic action commands known to the *soccer server*. At this abstraction level the skills correspond to simple actions such as turning towards a point. The middle layer contains intermediate skills which are based on low-level skills. The skills in this layer do not have to deal with the exact format of server messages anymore but can be specified in terms of the skills from the layer below. Finally, the skills at the highest level are based on intermediate skills. The way in which a skill is executed depends on the arguments which are supplied to it. The skills at each level receive one or more arguments of which the values depend on the current world state. The behavior of the agent is the result of selecting an appropriate skill in a given situation. Note that the agent is

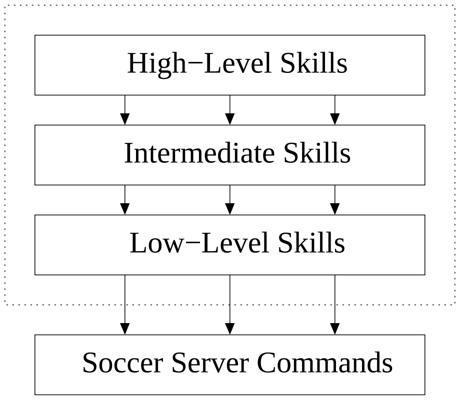


Figure 7.1: The *UvA Trilearn* skills hierarchy consisting of three layers. The low-level skills are based on *soccer server* commands, whereas the higher-level skills are based on skills from the layer below.

allowed to select skills from each layer and is not restricted to selecting high-level skills only. Which skill is selected in a certain situation depends on the team strategy and will be discussed in a later chapter.

It is important to realize that the execution of each skill eventually leads to a basic *soccer server* action command. For low-level player skills this mapping is clear since these can be directly specified in terms of basic action commands. At the higher levels however, the skills are implemented in terms of various skills from the level below and it then depends on the current situation which of these lower-level skills has to be executed. For some high-level skills this choice depends on several configurable threshold parameters which will be introduced later in this chapter. In our current implementation, the values for these parameters have been based on test results and on observations made during practice games. Note that selecting a skill will in itself only lead to the generation of a corresponding action command and not directly to its execution. This makes it possible to adapt the command parameters at a later stage when necessary. This can happen, for example, when an exhausted agent wants to move to his strategic position and has to perform a **dash** command with full power in order to achieve this. Before executing the command, the dash power will then be reduced thereby taking into account the agent's current stamina (see Section 9.5.3). Furthermore note that an agent will always commit to only a single action during each cycle regardless of which skill has been selected. The agent thus makes no commitment to a previous 'plan', e.g. if he executed part of a ball-interception skill in the previous cycle there is no guarantee that he will continue to execute the next part in the current cycle. Instead, the situation is completely reevaluated before each action opportunity and a skill is selected based on the current situation. This form of coordination between reasoning and execution is referred to as weak binding [116] (see Section 4.4).

Throughout this chapter a detailed description of each available skill will be presented in the form of equations and algorithms which use the information and methods available from the world model or one of the utility classes (see Section 4.3). For this we will use the following notation:

- (p_x^t, p_y^t): the agent's global position in cycle t. This will also be denoted by $\vec{p}_t$. Global positions of other objects will generally be referred to as (q_x^t, q_y^t) or $\vec{q}_t$. Relative positions are indicated as $\tilde{q}_t$.
- (v_x^t, v_y^t): the velocity of the agent in cycle t. This will also be denoted by $\vec{v}_t$. Velocities of other objects will generally be referred to as (w_x^t, w_y^t) or $\vec{w}_t$.
- θ^t: the agent's global neck angle in cycle t.

- ϕ^t: the angle of the agent's body relative to his neck in cycle t. Note that the agent's global body angle thus equals $\theta^t + \phi^t$.
- $\bar{x}^{t+i}$: the predicted value of x in cycle $t+i$. Here x can be either a position (p for the agent, q for other objects), a velocity (v for the agent, w for other objects), the agent's global neck angle θ or the agent's body angle ϕ relative to his neck.
- $\chi(x^t, cmd)$: a method that returns the predicted value of x in cycle $t+1$ given the value for x in cycle t after performing the action command cmd. These methods were described in Section 6.5.
- (v_r, v_ϕ): the vector $\vec{v}$ represented in polar coordinates, i.e. v_r denotes the length of the vector and v_ϕ its global direction.
- $\pi(r, \phi)$: a method that converts polar to Cartesian coordinates as shown in Equation 6.14.
- $\rho(v, \alpha)$: a method that rotates the vector v over α degrees.
- $\nu(\alpha)$: a method that converts a supplied angle α to an equivalent angle from the interval $[-180, 180]$. This will be referred to as the 'normalization' of the angle.
- $\tau(\alpha)$: a method that returns the angle value that must be supplied to a **turn** command in order to turn a player's body by a desired angle α. This method τ corresponds to the `getAngleForTurn` method which has been described in Section 6.6. The turn parameter is computed according to Equation 6.58 taking into account the speed and inertia of the player. Note that the return value will never exceed the maximum turn angle and will thus always come from the interval $[-180, 180]$.
- $\delta(\tilde{q})$: a method that returns the power that must be supplied to a **dash** command in order to come as close to a desired relative position $\tilde{q}$ as possible. This method δ corresponds to the `getPowerForDash` method which has been described in Section 6.6. The dash parameter is calculated according to Equation 6.63 taking into account the velocity and effort value of the agent. Since the agent can only move forwards or backwards, the closest point to the target position that he can reach by dashing is the closest point to $\tilde{q}$ that lies in the continuation of his body direction. Note that the return value will never exceed the maximum dash power as was explained in Section 6.6.
- $\lambda()$: a method that returns the actual kick power rate in a given situation. This method λ corresponds to the `getActualKickPowerRate` method which has been described in Section 6.6. The actual kick power rate is computed according to Equation 6.59 taking into account the position of the ball relative to the agent. When this value is multiplied by the actual kick power on a kick this gives the size of the resulting acceleration vector.
- $\kappa(s)$: a method that returns the power that must be supplied to a **kick** command in order to give a desired speed s to the ball on a kick. This method κ corresponds to the `getKickPowerForSpeed` method which has been described in Section 6.6. The kick parameter is determined according to Equation 6.60 taking the actual kick power rate into account. Note that the return value can exceed the maximum kick power. This makes it possible to identify situations in which the desired speed cannot be reached (see Sections 7.2.8 and 7.3.5).
- $\gamma(d, e)$: a method that returns the initial speed that has to be given to the ball on a kick such that it has a remaining speed equal to e after traveling distance d. This method γ corresponds to the `getKickSpeedToTravel` method which has been described in Section 6.6. The initial speed is calculated according to Equation 6.62 taking into account the velocity decay rate of the ball.
- $\psi(\alpha_{min}, \alpha_{max}, d)$: a method that returns the global direction α from the interval $[\alpha_{min}, \alpha_{max}]$ of the bisector of the widest angle between opponents that are located within distance d from the ball. This method ψ corresponds to the `getDirectionOfWidestAngleBetweenOpponents` method which has been described in Section 6.6.

- $\mu(\alpha_1, ..., \alpha_n)$: a method that returns the average of a given list of angles $[\alpha_1, \ldots, \alpha_n]$. Note that computing the average of a list of angles is not trivial due to the fact that angles go from 0 to 360 degrees and then back to 0 again. The average of 1 degree and 359 degrees should thus be 0 and not 180. This 'wrap around' problem can be solved by calculating the average of the sines of the angles as well as the average of the cosines of the angles. The true average angle can then be reconstructed by computing the arc tangent of the quotient of these respective averages. The method μ is thus implemented according to the following formula:

$$\mu(\alpha_1, ..., \alpha_n) \quad = \quad \nu(\text{atan2}(\frac{1}{n}\sum_{i=1}^{n}\sin(\alpha_i), \frac{1}{n}\sum_{i=1}^{n}\cos(\alpha_i))) \tag{7.1}$$

  where 'atan2(x,y)' is a function that computes the value of the arc tangent of x/y using the signs of both arguments to determine the quadrant of the return value and where ν converts the resulting angle to an equivalent angle from the interval $[-180, 180]$ as defined above.

7.2 Low-level Player Skills

Low-level player skills correspond to simple individual actions which can be directly specified in terms of basic action commands known to the *soccer server*. The way in which these actions are executed depends on the current state of the agent and its environment. In this section we describe the low-level player skills which are available to the *UvA Trilearn* agents. Each of these skills returns a basic action command of which the correct parameter values depend on the current world state. Since there exists a direct mapping between low-level player skills and basic action commands, these skills do not contain any complex decision procedures. Instead, their implementation is completely based on the known *soccer server* specifications.

7.2.1 Aligning the Neck with the Body

This skill enables an agent to align his neck with his body. It returns a **turn_neck** command that takes the angle ϕ of the agent's body relative to his neck as its only argument. Note that the angle ϕ can be directly extracted from the agent's world model. This leads to the following definition for this skill:

$$\texttt{alignNeckWithBody()} \quad = \quad (\textbf{turn\_neck} \;\; \phi^t) \tag{7.2}$$

7.2.2 Turning the Body towards a Point

This skill enables an agent to turn his body towards a given point. It receives a global position $\vec{q}$ on the field and returns a **turn** command that will turn the agent's body towards this point. To this end the agent's global position $\bar{p}^{t+1}$ in the next cycle is predicted based on his current velocity. This is done to compensate for the fact that the remaining velocity will move the agent to another position in the next cycle. The global angle $(\vec{q} - \bar{p}^{t+1})_\phi$ between the given position and the predicted position is then determined after which the agent's global body direction $\theta^t + \phi^t$ is subtracted from this angle in order to make it relative to the agent's body. Finally, the resulting angle is normalized and supplied as an argument to the τ method in order to compute the angle value that must be passed to the **turn** command. If the speed and inertia of the agent make it impossible to turn towards the given position in a single cycle then the agent turns as far as possible. This leads to the following definition for this skill:

$$\texttt{turnBodyToPoint}(\vec{q}) \quad = \quad (\textbf{turn} \;\; \tau(\nu((\vec{q} - \bar{p}^{t+1})_\phi - (\theta^t + \phi^t)))) \tag{7.3}$$

7.2.3 Turning the Back towards a Point

This skill enables an agent to turn his back towards a given point $\vec{q}$. The only difference between this skill and `turnBodyToPoint` is that the angle $(\vec{q} - \bar{p}^{t+1})_\phi$ between the given position and the predicted position of the agent in the next cycle is now made relative to the back of the agent by subtracting the agent's global back direction $\theta^t + \phi^t + 180$. This skill is used by the goalkeeper in case he wants to move back to his goal while keeping sight of the rest of the field. It can be defined as follows:

$$\texttt{turnBackToPoint}(\vec{q}) \quad = \quad (\textbf{turn} \;\; \tau(\nu((\vec{q} - \bar{p}^{t+1})_\phi - (\theta^t + \phi^t + 180)))) \tag{7.4}$$

7.2.4 Turning the Neck towards a Point

This skill enables an agent to turn his neck towards a given point. It receives a global position $\vec{q}$ on the field as well as a primary action command *cmd* that will be executed by the agent at the end of the current cycle and returns a **turn_neck** command that will turn the agent's neck towards $\vec{q}$. To this end, the agent's global position and neck direction after executing the *cmd* command are predicted using methods from the world model. The global angle between the given position and the predicted position is then determined after which the predicted neck direction is subtracted from this angle in order to make it relative to the agent's neck. Finally, the resulting angle is normalized and directly passed as an argument to the **turn_neck** command since the actual angle by which a player turns his neck is by definition equal to this argument (see Section 3.4.5). If the resulting turn angle causes the absolute angle between the agent's neck and body to exceed the maximum value, then the agent turns his neck as far as possible. Note that it is necessary to supply the selected primary command as an argument to this skill, since a **turn_neck** command can be executed in the same cycle as a **kick**, **dash**, **turn**, **move** or **catch** command (see Section 3.4.9). Turning the neck towards a point can be defined as follows:

$$\texttt{turnNeckToPoint}(\vec{q}, cmd) \quad = \quad (\textbf{turn\_neck} \;\; \nu((\vec{q} - \chi(\vec{p}_t, cmd))_\phi - \chi(\theta^t, cmd))) \tag{7.5}$$

7.2.5 Searching for the Ball

This skill enables an agent to search for the ball when he cannot see it. It returns a **turn** command that causes the agent to turn his body by an angle that equals the width of his current view cone (denoted by the *ViewAngle* attribute in the *AgentObject* class; see Figure 6.2). In this way the agent will see an entirely different part of the field after the **turn** which maximizes the chance that he will see the ball in the next cycle. Note that the agent always turns towards the direction in which the ball was last observed to avoid turning back and forth without ever seeing the ball. Let *Sign* be equal to $+1$ when the relative angle to the point at which the ball was last observed is positive and -1 otherwise. The angle value that must be supplied to the **turn** command in order to turn the desired angle can then be computed by supplying $(Sign \cdot ViewAngle)$ as an argument to the τ method. This leads to the following definition for this skill:

$$\texttt{searchBall()} \quad = \quad (\textbf{turn} \;\; \tau(Sign \cdot ViewAngle)) \tag{7.6}$$

7.2.6 Dashing to a Point

This skill enables an agent to dash to a given point. It receives a global position $\vec{q}$ as its only argument and returns a **dash** command that causes the agent to come as close to this point as possible. Since the agent can only move forwards or backwards, the closest point to the target position that he can reach by dashing is the orthogonal projection of $\vec{q}$ onto the line that extends into the direction of his body (forwards and backwards). The power that must be supplied to the **dash** command is computed using the δ method which takes the position of $\vec{q}$ relative to the agent as input. This skill can then be defined as follows:

$$\texttt{dashToPoint}(\vec{q}) \quad = \quad (\textbf{dash} \;\; \delta(\vec{q} - \vec{p}_t)) \tag{7.7}$$

7.2.7 Freezing the Ball

This skill enables an agent to freeze a moving ball, i.e. it returns a **kick** command that stops the ball dead at its current position. Since ball movement in the *soccer server* is implemented as a vector addition (see Equation 3.20), the ball will stop in the next cycle when it is kicked in such a way that the resulting acceleration vector has the same length and opposite direction to the current ball velocity. The desired speed that should be given to the ball on the kick thus equals the current ball speed w_r^t and the power that must be supplied to the **kick** command to achieve this can be computed using the κ method. Furthermore, the direction of the kick should equal the direction w_ϕ^t of the current ball velocity plus 180 degrees. Note that this direction must be made relative to the agent's global body angle $\theta^t + \phi^t$ before it can be passed as an argument to the **kick** command. The following definition can be given for this skill:

$$\texttt{freezeBall()} \quad = \quad (\textbf{kick} \;\; \kappa(w_r^t) \;\; \nu((w_\phi^t + 180) - (\theta^t + \phi^t))) \tag{7.8}$$

7.2.8 Kicking the Ball Close to the Body

This skill enables an agent to kick the ball close to his body. It receives an angle α as its only argument and returns a **kick** command that causes the ball to move to a point at a relative angle of α degrees and at a close distance (`kickable_margin`/6 to be precise) from the agent's body. To this end the ball has to be kicked from its current position $\vec{q}_t$ to the desired point relative to the predicted position $\vec{p}^{t+1}$ of the agent in the next cycle[1]. An example situation is shown in Figure 7.2. In general, this skill will be used when the agent wants to kick the ball to a certain position on the field which cannot be reached with a single kick. Since the efficiency of a kick is highest when the ball is positioned just in front of the agent's body (see Section 3.4.1), calling this skill with $\alpha = 0$ will have the effect that the agent can kick the ball with more power after it is executed. In order to get the ball to the desired point in the next cycle, the ball movement vector $\vec{u}_{t+1}$ must be equal to:

$$\vec{u}_{t+1} \quad = \quad \vec{p}^{t+1} + \pi(\texttt{player\_size} + \texttt{ball\_size} + \texttt{kickable\_margin}/6, \nu(\theta^t + \phi^t + \alpha)) - \vec{q}_t \tag{7.9}$$

It follows from Equation 3.20 that the acceleration vector $\vec{a}_t$ resulting from the kick should then be:

$$\vec{a}_t \quad = \quad \vec{u}_{t+1} - \vec{w}_t \tag{7.10}$$

where $\vec{w}_t$ denotes the current ball velocity. The power that must be supplied to the **kick** command to achieve the desired acceleration can be computed by passing the length a_r^t of the acceleration vector as an argument to the κ method. The direction of the kick should equal the direction a_ϕ^t of the acceleration vector relative to the agent's global body angle $\theta^t + \phi^t$. This leads to the following definition for this skill:

$$\texttt{kickBallCloseToBody}(\alpha) \quad = \quad (\textbf{kick} \;\; \kappa(a_r^t) \;\; \nu((a_\phi^t - (\theta^t + \phi^t)))) \tag{7.11}$$

Note that this skill will only be executed if it is possible to actually reach the desired ball position with a single kick, i.e. if the return value of the κ method is smaller than or equal to the maximum kick power. If the required power does exceed the maximum then the ball is frozen at its current position using the `freezeBall` skill described in Section 7.2.7. In general, it will then always be possible to shoot the motionless ball to the desired point in the next cycle.

[1] Note that the ball can go 'through' the agent as a result of the fact that positions are only updated at the end of each cycle.

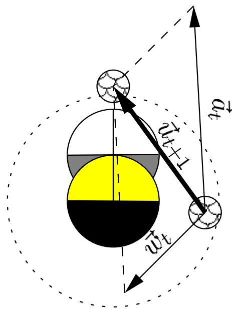


Figure 7.2: Example situation for the `kickBallCloseToBody` skill with $\alpha = 0$. The acceleration vector $\vec{a}_t$ takes the current ball velocity $\vec{w}_t$ into account to produce the movement vector $\vec{u}_{t+1}$. As a result, the kick moves the ball from its current position to a position just in front of the predicted position of the agent in the next cycle. The dotted circle indicates the agent's kickable range in cycle t.

7.2.9 Accelerating the Ball to a Certain Velocity

This skill enables an agent to accelerate the ball in such a way that it gets a certain velocity after the kick. It receives the desired velocity $\vec{w}_d$ as its only argument and returns a **kick** command that causes the ball to be accelerated to this velocity. The acceleration vector $\vec{a}_t$ that is required to achieve this equals:

$$\vec{a}_t \quad = \quad \vec{w}_d - \vec{w}_t \tag{7.12}$$

where $\vec{w}_t$ denotes the current ball velocity. If the power $\kappa(a_r^t)$ that must be supplied to the **kick** command to get the desired result does not exceed the maximum kick power then the desired velocity can be realized with a single kick. The kick direction should then be equal to the direction a_ϕ^t of the acceleration vector relative to the agent's global body angle $\theta^t + \phi^t$. This leads to the following definition for this skill:

$$\texttt{accelerateBallToVelocity}(\vec{w}_d) \quad = \quad (\textbf{kick} \;\; \kappa(a_r^t) \;\; \nu(a_\phi^t - (\theta^t + \phi^t))) \tag{7.13}$$

However, if the desired velocity is too great or if the current ball velocity is too high then the required acceleration cannot be realized with a single kick. In this case, the ball is kicked in such a way that the acceleration vector has the maximum possible length and a direction that aligns the resulting ball movement with $\vec{w}_d$. This means that after the kick the ball will move in the same direction as $\vec{w}_d$ but at a lower speed. To this end, the acceleration vector has to compensate for the current ball velocity in the 'wrong' direction (y-component). This gives the following formula for computing the acceleration vector:

$$\vec{a}_t \quad = \quad \pi(\lambda() \cdot \texttt{maxpower}, \, \nu((\vec{w}_d)_\phi - \text{asin}(\frac{\rho(\vec{w}_t, -(\vec{w}_d)_\phi))_y}{\lambda() \cdot \texttt{maxpower}})) \tag{7.14}$$

The skill can then be defined as follows:

$$\texttt{accelerateBallToVelocity}(\vec{w}_d) \quad = \quad (\textbf{kick} \;\; \texttt{maxpower} \;\; \nu(a_\phi^t - (\theta^t + \phi^t))) \tag{7.15}$$

7.2.10 Catching the Ball

This skill enables an agent to catch the ball and can only be executed when the agent is a goalkeeper. It returns a **catch** command that takes the angle of the ball relative to the body of the agent as its only argument. The correct value for this argument is computed by determining the global direction between the current ball position $\vec{q}_t$ and the agent's current position $\vec{p}_t$ and by making this direction relative to the agent's global body angle $\theta^t + \phi^t$. This skill can be expressed by the following definition:

$$\texttt{catchBall} = (\textbf{catch}\ \ \nu((\vec{q}_t - \vec{p}_t)_\phi - (\theta^t + \phi^t))) \tag{7.16}$$

7.2.11 Communicating a Message

This skill enables an agent to communicate with other players on the field. It receives a string message m as its only argument and returns a **say** command that causes the message to be broadcast to all players within a certain distance from the speaker. This skill can be defined as follows:

$$\texttt{communicate}(m) = (\textbf{say}\ \ m) \tag{7.17}$$

7.3 Intermediate Player Skills

Intermediate player skills correspond to actions at a higher abstraction level than those presented in Section 7.2. The skills at this level do not have to deal with the exact format of server messages anymore but use the low-level player skills to generate the desired behavior. In this section we describe the intermediate player skills which are available to the *UvA Trilearn* agents. All of these skills eventually lead to the execution of a basic action command which is generated by a skill from the layer below. In its simplest form, an intermediate skill uses the current state of the environment to determine the correct argument values with which a specific low-level skill has to be called. In most cases however, they also contain a simple decision procedure for selecting among a small number of related skills. Note that the intermediate skills themselves form parts of more complex high-level skills which will be discussed in Section 7.4.

7.3.1 Turning the Body towards an Object

This skill enables an agent to turn his body towards an object o which is supplied to it as an argument. To this end, the object's global position $\bar{q}_o^{t+1}$ in the next cycle is predicted based on its current velocity. This predicted position is passed as an argument to the `turnBodyToPoint` skill which generates a **turn** command that causes the agent to turn his body towards the object. This leads to the following definition:

$$\texttt{turnBodyToObject}(o) = \texttt{turnBodyToPoint}(\bar{q}_o^{t+1}) \tag{7.18}$$

7.3.2 Turning the Neck towards an Object

This skill enables an agent to turn his neck towards an object. It receives as arguments this object o as well as a primary action command *cmd* that will be executed by the agent at the end of the current cycle. Turning the neck towards an object amounts to predicting the object's global position $\bar{q}_o^{t+1}$ in the next cycle and passing this predicted position together with the *cmd* command as arguments to the `turnNeckToPoint` skill. This low-level skill will then generate a **turn_neck** command that causes the agent to turn his neck towards the given object. Note that the *cmd* command is supplied as an argument for predicting the agent's global position and neck angle after executing the command. This is necessary because a **turn_neck** command can be executed in the same cycle as a **kick**, **dash**, **turn**, **move** or **catch** command (see Section 3.4.9). The following definition can be given for this skill:

$$\texttt{turnNeckToObject}(o, cmd) = \texttt{turnNeckToPoint}(\bar{q}_o^{t+1}, cmd) \tag{7.19}$$

7.3.3 Moving to a Position

This skill enables an agent to move to a global position $\vec{q}$ on the field which is supplied to it as an argument. Since the agent can only move forwards or backwards into the direction of his body, the crucial decision in the execution of this skill is whether he should perform a **turn** or a **dash**. Turning has the advantage that in the next cycle the agent will be orientated correctly towards the point he wants to reach. However, it has the disadvantage that performing the **turn** will cost a cycle and will reduce the agent's velocity since no acceleration vector is added in that cycle[2]. Apart from the target position $\vec{q}$, this skill receives several additional arguments for determining whether a **turn** or **dash** should be performed in the current situation. If the target point is in front of the agent then a **dash** is performed when the relative angle to this point is smaller than a given angle α. However, if the target point is behind the agent then a **dash** is only performed if the distance to this point is less than a given value d and if the angle relative to the back direction of the agent is smaller than α. In all other cases a **turn** is performed. Figure 7.3 shows in which situations the agent will **turn** or **dash** depending on the position of the target point.

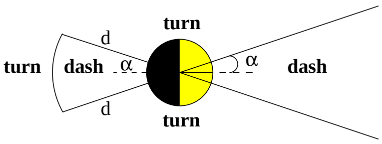


Figure 7.3: Situations in which the agent turns or dashes in the `moveToPos` skill assuming $b = \mathit{false}$.

Note that in the case of the goalkeeper it is sometimes desirable that he moves backwards towards his goal in order to keep sight of the rest of the field. To this end an additional boolean argument b is supplied to this skill that indicates whether the agent should always move backwards to the target point. If b equals *true* then the agent will turn his back towards the target point if the angle relative to his back direction is larger than α. In all other cases, he will perform a (backward) **dash** towards $\vec{q}$ regardless of whether the distance to this point is larger than d. A complete definition for this skill is given in Algorithm 7.1.

<u>`moveToPos`$(\vec{q}, \alpha, d, b)$</u>

ang $= \nu((\vec{q} - \vec{p}_t)_\phi - (\theta^t + \phi^t))$, dist $= (\vec{q} - \vec{p}_t)_r$
if b == *true* **then**
  ang $= \nu(\text{ang} + 180)$ // always turn back to target point
  if $|\text{ang}| < \alpha$ **then**
    return `dashToPoint`$(\vec{q})$
  else
    return `turnBackToPoint`$(\vec{q})$
  end if
else
  if $|\text{ang}| < \alpha$ **or** $(|\nu(\text{ang} + 180)| < \alpha$ **and** dist $< d)$ **then**
    return `dashToPoint`$(\vec{q})$
  else
    return `turnBodyToPoint`$(\vec{q})$
  end if
end if

Algorithm 7.1: Pseudo-code implementation for moving to a desired position.

[2] Recall from Section 3.4.2 that the agent accelerates to maximum speed by executing a sequence of **dash** commands. Breaking that sequence will slow him down after which he will need a few cycles to reach maximum speed again.

7.3.4 Intercepting a Close Ball

This skill enables an agent to intercept a ball which is close to him. The objective is to move in such a way that the ball will come within a small distance d from the agent in one or two cycles. To this end, the prediction methods from the world model are used to predict the ball position in the next cycle and two cycles into the future. It is then determined whether it is possible to move the agent within distance d from one of these positions using all logical combinations of **turn** and **dash** commands. If it is not possible to intercept the ball within two cycles then this skill returns an illegal command to indicate that it cannot be performed. Note that the distance d is supplied as an argument to this skill. For field players it is equal to the kickable distance, whereas for the goalkeeper the catchable distance is used.

First it is determined whether the agent can intercept the ball in one cycle. To this end, the position $\bar{q}^{t+1}$ of the ball in the next cycle is predicted and a calculation is performed to decide whether a single **dash** can move the agent within distance d from this position. In order to be able to kick the ball efficiently after intercepting it, it is important that the agent moves to a good position relative to the ball (i.e. the ball must be in front of him). At the same time the agent must make sure that he does not collide with the ball when trying to intercept it. Let l be a line that runs forwards and backwards from the predicted position $\bar{p}^{t+1}$ of the agent in the next cycle into the direction of his body. This line thus denotes the possible movement direction of the agent. Note that we have to use the agent's predicted position in the next cycle since his current velocity must be taken into account. In addition, let c be a circle which is centered on the predicted ball position $\bar{q}^{t+1}$ and which has a radius equal to the sum of the radius of the agent, the radius of the ball and a small buffer (`kickable_margin`/6). It is now determined whether the agent can intercept the ball in the next cycle by looking at the number of intersection points between l and c. If l and c have exactly one point in common then this point is the desired interception point for the next cycle. However, if the number of intersection points equals two then the desired point is the one for which the absolute angle of the ball relative to that point is the smallest. This amounts to the intersection point which is closest to the agent when the ball lies in front of him and to the furthest one when the ball is behind his back. As a result, the desired interception point will always be such that the agent has the ball in front of him in the next cycle. An example situation is shown in Figure 7.4. The `dashToPoint` skill is then used to generate a **dash** command that will bring the agent as close as possible to the desired point. Next, the position of the agent after executing this command is predicted and if it turns out that this predicted position lies within distance d from the ball then the **dash** is performed. However, if the predicted position is not close enough to the ball or if l and c have no points in common then it is assumed that the ball cannot be intercepted with a single **dash**. In these cases, two alternatives are explored to see if the ball can be intercepted in two cycles.

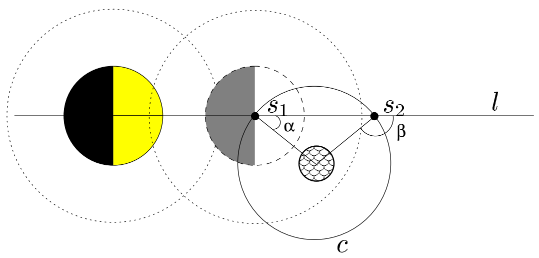


Figure 7.4: Example situation for intercepting a close ball in one cycle. The agent moves along the line l to one of the intersection points between l and c. In this case s_1 is the desired interception point since there the ball lies in front of the agent ($\alpha < \beta$). The dotted circles indicate the agent's kickable range.

The first alternative is to determine whether the agent can intercept the ball by performing a **turn** followed by a **dash**. To this end, the global position $\bar{q}^{t+2}$ of the ball is predicted two cycles into the future and the `turnBodyToPoint` skill is used to generate a **turn** command that will turn the agent towards this point. The agent's position $\bar{p}^{t+1}$ after executing this command is then predicted after which the `dashToPoint` skill is used to generate a **dash** command that will bring the agent as close as possible to $\bar{q}^{t+2}$. If it turns out that the predicted position $\bar{p}^{t+2}$ of the agent after the **dash** lies within distance d from the ball then the first command (i.e. the **turn**) in the sequence of two is performed. Otherwise, a second alternative is tried to determine whether the agent can intercept the ball by performing two **dash** commands. To this end the `dashToPoint` skill is used twice with the predicted ball position $\bar{q}^{t+2}$ after two cycles as its argument on both occasions. If the predicted position $\bar{p}^{t+2}$ of the agent after these two dashes lies within distance d from the ball then the first **dash** is performed. Otherwise, an illegal command is returned to indicate that the skill cannot be performed. The close interception procedure is largely based on a similar method introduced in [91]. A complete definition for this skill is given in Algorithm 7.2.

<u>`closeIntercept`(d)</u>

{try to intercept the ball in one cycle with a single dash}
l = line that goes through $\bar{p}^{t+1}$ into direction $(\bar{\theta}^{t+1} + \bar{\phi}^{t+1})$
c = circle centered on $\bar{q}^{t+1}$ with radius `player_size` + `ball_size` + `kickable_margin/6`.
if number of intersection points between l and c is greater than zero **then**
  determine intersection point s_i for which $|(\bar{q}^{t+1} - s_i)_\phi - (\bar{\theta}^{t+1} + \bar{\phi}^{t+1})|$ is minimal.
  if $(\chi(p^t, \texttt{dashToPoint}(s_i)) - \bar{q}^{t+1})_r < d$ **then**
    return `dashToPoint`(s_i)
  end if
end if

{try to intercept the ball in two cycles with a turn followed by a dash}
$\bar{p}^{t+1} = \chi(p^t, \texttt{turnBodyToPoint}(\bar{q}^{t+2}))$
$\bar{p}^{t+2} = \chi(\bar{p}^{t+1}, \texttt{dashToPoint}(\bar{q}^{t+2}))$
if $(\bar{p}^{t+2} - \bar{q}^{t+2})_r < d$ **then**
  return `turnBodyToPoint`($\bar{q}^{t+2}$)
end if

{try to intercept the ball in two cycles with two dashes}
$\bar{p}^{t+1} = \chi(p^t, \texttt{dashToPoint}(\bar{q}^{t+2}))$
$\bar{p}^{t+2} = \chi(\bar{p}^{t+1}, \texttt{dashToPoint}(\bar{q}^{t+2}))$
if $(\bar{p}^{t+2} - \bar{q}^{t+2})_r < d$ **then**
  return `dashToPoint`($\bar{q}^{t+2}$)
end if

return `CMD_ILLEGAL`

Algorithm 7.2: Pseudo-code implementation for intercepting a close ball.

7.3.5 Kicking the Ball to a Point at a Certain Speed

This skill enables an agent to kick the ball from its current position $\vec{o}_t$ to a given position $\vec{q}$ in such a way that it has a remaining speed equal to e when it reaches this position. In order for the ball to reach the target position at the desired speed, the ball velocity $\vec{w}_d$ after executing the **kick** must be equal to

$$\vec{w}_d = \pi(\gamma((\vec{q} - \vec{o}_t)_r, e), (\vec{q} - \vec{o}_t)_\phi) \tag{7.20}$$

However, it is possible that the ball cannot reach this velocity with a single **kick** either because the magnitude of $\vec{w}_d$ exceeds the maximum speed of the ball or due to the fact that the current ball speed in combination with the position of the ball relative to the agent make it impossible to achieve the required acceleration. If the magnitude of $\vec{w}_d$ is larger than `ball_speed_max` it is certain that even in the optimal situation (i.e. if the ball lies directly in front of the agent and has zero velocity) the agent will not be able to kick the ball to the target position at the desired speed. In this case, the expected ball movement $\bar{u}^{t+1}$ is computed after executing a **kick** with maximum power into the same direction as $\vec{w}_d$. If the magnitude of the resulting movement vector is larger than a given percentage h of the maximum ball speed[3] then this kick is actually performed despite the fact that it cannot produce the wanted effect. Otherwise, the agent shoots the ball close to his body and directly in front of him using the `kickBallCloseToBody` skill. In this way he will be able to kick the ball with more power in the next cycle. However, if the magnitude of $\vec{w}_d$ is smaller than `ball_speed_max` it is possible to reach the target point at the desired speed in the optimal situation. The acceleration vector $\vec{a}_t$ that is required to achieve this is then equal to

$$\vec{a}_t \quad = \quad \vec{w}_d - \vec{w}_t \tag{7.21}$$

where $\vec{w}_t$ denotes the current ball velocity. If the power that must be supplied to the **kick** command to achieve this acceleration is less than or equal to the maximum power then the `accelerateBallToVelocity` skill is used to perform the desired kick. Otherwise, the agent uses the `kickBallCloseToBody` skill to put the ball in a better kicking position for the next cycle. Figure 7.5 shows an example situation in which the `kickTo` skill is called in two consecutive cycles. In cycle t it is not possible to shoot the ball to the point q at the desired speed due to the fact that the ball has too much speed in the wrong direction. The agent therefore shoots the ball to a point just in front of him. In the next cycle he is then able to perform the desired kick. A complete definition for the `kickTo` skill is given in Algorithm 7.3.

<u>kickTo(q, e, h)</u>

$\vec{w}_d = \pi(\gamma((\vec{q} - \vec{o}_t)_r, e), (\vec{q} - \vec{o}_t)_\phi)$ // this is the desired ball velocity after the kick
if $(\vec{w}_d)_r >$ `ball_speed_max` **then**
  $\vec{a}_t = \pi(\lambda() \cdot \texttt{maxpower}, \nu((\vec{w}_d)_\phi - \text{asin}(\frac{\rho(\vec{w}_t, -(\vec{w}_d)_\phi)_y}{\lambda() \cdot \texttt{max\_power}})))$ // see Equation 7.14
  $\bar{u}^{t+1} = \vec{a}_t + \vec{w}_t$
  if $(\bar{u}^{t+1})_r > h \cdot$ `ball_speed_max` **then**
    return `accelerateBallToVelocity`($\vec{w}_d$) // shoot with maximum power in same direction as $\vec{w}_d$
  else
    return `kickBallCloseToBody`(0) // put the ball in a better position
  end if
else
  $\vec{a}_t = \vec{w}_d - \vec{w}_t$ // this is the required acceleration vector
  if $\kappa(a_r^t) <$ `maxpower` **then**
    return `accelerateBallToVelocity`($\vec{w}_d$) // target can be reached
  else
    return `kickBallCloseToBody`(0) // put the ball in a better position
  end if
end if

Algorithm 7.3: Pseudo-code implementation for kicking the ball to a desired point at a certain speed.

[3] In our current implementation h is usually equal to the threshold parameter *KickMaxThr* which has a value of 0.85.

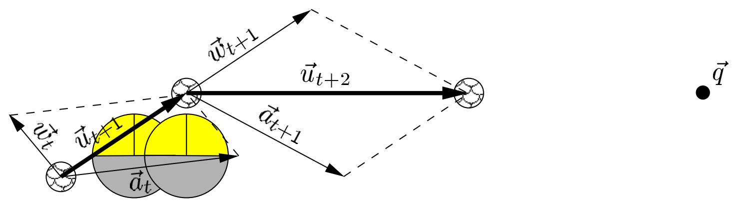


Figure 7.5: Example situation in which the `kickTo` skill is called in two consecutive cycles. In cycle t the agent kicks the ball in front of his body causing the ball movement $\vec{u}_{t+1}$. In cycle $t+1$ he is then able to kick the ball towards $\vec{q}$ at the desired speed. Note that in this example the position of the agent changes from cycle t to cycle $t+1$ due to his remaining velocity.

7.3.6 Turning with the Ball

This skill enables an agent to turn towards a global angle α while keeping the ball in front of him. It is used, for example, when a defender has intercepted the ball in his defensive area and faces his own goal. In this situation, the defender usually wants to pass the ball up the field into an area that is currently not visible to him and to this end he will first use this skill to turn with the ball towards the opponent's goal. Turning with the ball requires a sequence of commands to be performed. The ball first has to be kicked to a desired position relative to the agent, then it has to be stopped dead at that position and finally the agent must turn towards the ball again. Each time when this skill is called it has to be determined which part of the sequence still has to be executed[4]. This is done as follows. If the absolute difference between the desired angle α and the global angle $(\vec{q}_t - \vec{p}_t)_\phi$ of the ball relative to the position of the agent is larger than a given value k then the `kickBallCloseToBody` skill is used to kick the ball to a position close to the agent and at the desired angle. Otherwise, it is checked whether the ball still has speed from the previous action. If the remaining ball speed w_r^t exceeds a given value s then the ball is stopped dead at its current position using the `freezeBall` skill. In all other cases the agent turns his body towards the predicted position $\bar{q}^{t+1}$ of the ball in the next cycle. Note that in our current implementation k and s are usually equal to the threshold parameters *TurnWithBallAngle* and *TurnWithBallSpeed* which have respective values of 30 and 0.1. A complete definition for this skill is given in Algorithm 7.4.

<u>`turnWithBallTo`(α, k, s)</u>

if $|(\nu((\vec{q}_t - \vec{p}_t)_\phi - \alpha)| > k$ **then**
  return `kickBallCloseToBody`$(\nu(\alpha - (\theta^t + \phi^t)))$
else if $w_r^t > s$ **then**
  return `freezeBall`$()$
else
  return `turnBodyToPoint`$(\bar{q}^{t+1})$
end if

Algorithm 7.4: Pseudo-code implementation for turning with the ball.

[4]Note that it is not always the case that the sequence will be finished. It is possible that during the sequence some external event (e.g. an opponent coming close) causes the agent to decide upon another action (e.g. passing to a teammate).

7.3.7 Moving to a Position While Staying on a Line

This skill enables an agent to move along a line l to a given position $\vec{q}$ on this line. It is used, for example, by the goalkeeper who often wants to stay on a line in front of his goal and move to different positions on this line depending on where the ball is located. Furthermore, it can also used by defenders for marking an opponent player by moving along a line between this player and the ball. The idea is that the agent must try to move as fast as possible to the desired point $\vec{q}$ along the line l thereby keeping the number of turns to a minimum to avoid wasting cycles. Apart from the target position $\vec{q}$, this skill receives several additional arguments for determining whether the agent should **turn** or **dash** in the current situation. Since the agent can only move forwards or backwards into the direction of his body, it is important that he tries to keep the orientation of his body aligned with the direction of l in order to be able to move quickly to the target point. A given angle α denotes the desired body angle (global) of the agent in the point $\vec{q}$. The line l can thus be defined as going through $\vec{q}$ and having global direction α. Due to the noise that is added to the movement of the agent, the orientation of his body will never be exactly equal to α and as a result the agent's position will start to deviate from the line. Each time when this skill is called, the agent's desired orientation α is therefore slightly adjusted depending on his position with respect to l. If the distance d between the agent's current position $\vec{p}$ and the line l is smaller than a given value h then α remains unchanged. However, if d exceeds h then α is adjusted in such a way that the agent will move closer to l in subsequent cycles. This is done by either increasing or decreasing the desired orientation α by α_{corr} degrees depending on which side of the line the agent is located and on a prediction of the agent's movement in the forthcoming cycles. This prediction is represented by a given value *sign* which equals 1 if the agent is expected to move in the same direction as α and -1 if he will move in the opposite direction. Adjusting α in this way has the effect that in subsequent cycles the agent will move closer to the line again if this is necessary. The final decision whether to **turn** or **dash** is now made by comparing the agent's current body angle to the desired orientation α. If the absolute difference between these two angles is larger than α_{thr} degrees then the agent uses the `turnBodyToPoint` skill to turn in the desired direction. Otherwise, the `dashToPoint` skill is called to move towards the target position.

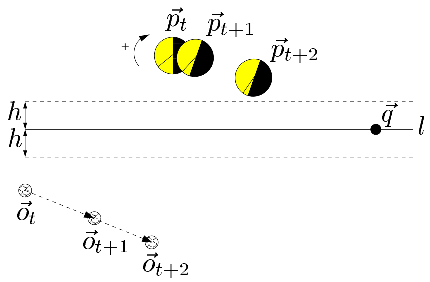


Figure 7.6: Example situation in which the `moveToPosAlongLine` skill is called in three consecutive cycles. The agent wants to move to the point $\vec{q}$ while keeping sight of the ball.

Figure 7.6 shows an example situation in which this skill is called in three consecutive cycles. The agent wants to move along the line l to the point $\vec{q}$ while keeping sight of the ball. The values for α and *sign* are equal to -180 degrees and -1 respectively, indicating that the agent's body should be turned towards the left of the picture when he reaches $\vec{q}$ and that his movement in the following cycles will be in the opposite direction. In cycle t the distance between the agent's position $\vec{p}_t$ and the line l is larger than h and the agent's body direction differs more than α_{thr} degrees from the desired orientation α. The agent therefore turns in positive (i.e. clockwise) direction to the corrected value of α. A backward dash in the next cycle then moves him closer to both l and $\vec{q}$. A complete definition for this skill is given in Algorithm 7.5.

```
moveToPosAlongLine(q⃗, α, h, sign, α_thr, α_corr)

l = line with global direction α that goes through q⃗
p⃗' = perpendicular projection of the agent's position p⃗_t onto l
dist = (p⃗_t − p⃗')_r
if dist > h then
  let m be the line through p⃗_t and the origin O and set s⃗ = intersection(m, l)
  if (p⃗_t − O)_r < (s⃗ − O)_r then
    side = 1                       // the agent stands between O and l
  else
    side = −1                      // l runs between O and p⃗_t
  end if
  α = α + sign · side · α_corr     // correct desired body angle
end if
if |ν(α − (θ^t + φ^t))| > α_thr then
  return turnBodyToPoint(p⃗_t + π(1.0, α))
else
  return dashToPoint(q⃗)
end if
```

Algorithm 7.5: Pseudo-code implementation for moving to a position along a line.

7.4 High-level Player Skills

High-level player skills correspond to actions at the highest abstraction level in the skills hierarchy depicted in Figure 7.1. The skills at this level use the intermediate player skills presented in Section 7.3 to generate the desired behavior. In this section we describe the high-level player skills which are available to the *UvA Trilearn* agents. These skills use the current state of the world to decide which intermediate skill has to be called and this eventually leads to the execution of a basic action command. In most cases, the correct intermediate skill is chosen based on a number of configurable threshold parameters which will be introduced throughout this section. In our current implementation the values for these parameters have been based on experimental results and on observations made during test matches. It is important to realize that high-level player skills cannot be regarded as strategic behaviors in themselves. Instead, they are called as a result of a strategic decision making procedure which will be discussed in Chapter 9.

7.4.1 Intercepting the Ball

This skill enables an agent to intercept a ball at any distance. Intercepting the ball is one of the most important available player skills that is frequently used by every type of player. If a player wants to kick the ball he must always intercept it first, i.e. get close to it. The main objective is to determine the optimal interception point based on the current position and velocity of the ball and to move to that point as fast as possible in order to reach the ball before an opponent does. This obviously requires a sequence of commands to be performed. Ball interception in the *soccer server* is not a straightforward task due to the noise that is added to sensory observations and to the movement of the ball. This noise makes it difficult to accurately predict the ball's future trajectory and thus the optimal interception point. However, once the interception point is determined it is important that it is a good estimate since the agent can only move forwards and backwards into the direction of his body. If during an intercept sequence it turns out that the chosen interception point is not well predicted then the agent is forced to turn towards a newly calculated point which will slow him down considerably. In [90] two possible methods are described for equipping simulated soccer agents with the ability to intercept a moving ball:

1. An *analytical* method in which the ball velocity is estimated using past positions and in which the future movement of the ball is predicted based on this velocity.
2. An *empirical* method in which a neural network is used to create a general ball-interception behavior based on examples of successful interceptions.

Statistics presented in [90] show that the empirical method using a neural network gives slightly better results than the analytical solution. It must be noted however, that the experiments performed to test these methods were performed in an old version of the *soccer server* (version 2) in which a player did not yet receive distance change and direction change information of dynamic objects and in which visual information was sent to the players at larger intervals making an accurate prediction of the ball's future trajectory even more difficult. The above-mentioned results can therefore not be generalized to later versions of the *soccer server*. It turns out that the results for the analytical method improve significantly when visual messages are received more frequently and when change information about mobile objects can be used to predict the velocity of the ball. The ball-interception skill used by the *UvA Trilearn* agents is therefore based on the analytical method presented in [90].

When the ball-interception skill is called, it is first determined whether it is possible for the agent to intercept the ball within two cycles using the intermediate player skill `closeIntercept`. When the agent is a field player this skill takes the agent's kickable range as its only argument, whereas for the goalkeeper the catchable range is supplied (see Section 7.3.4). If it turns out that the ball cannot be intercepted within two cycles then the agent uses an iterative scheme to compute the optimal interception point. A loop is executed in which the prediction methods described in Section 6.5 are used to predict the position $\bar{q}^{t+i}$ of the ball a number of cycles, say i, into the future and to predict the number of cycles, say n, that the agent will need to reach this position. This is repeated for increasing values of i until $n < i$ in which case it is assumed thay the agent should be able to reach the point $\bar{q}^{t+i}$ before the ball does. If at this point the value for i has become larger than *InterceptMaxCycles* (which has a value of 30 in our current implementation) an illegal command is returned to indicate that the agent should not try to intercept the ball. Otherwise, the point $\bar{q}^{t+i}$ is chosen as the interception point and the `moveToPos` skill is used to move towards this point. Recall from Section 7.3.3 that the decision whether to **turn** or **dash** in the current situation depends on the angle of the target point relative to the agent's body direction and on the distance to the target point if it lies behind the agent. In this case, the `moveToPos` skill uses the threshold parameters *InterceptTurnAngle* and *InterceptDistanceBack* to make this decision[5]. A pseudo-code implementation for the ball-interception skill is given in Algorithm 7.6.

[5] In our current implementation these parameters have respective values of 7 degrees and 2 meters.

```
intercept()

if agent == goalkeeper then
  dist = catchable_area_l
else
  dist = kickable_distance          // = player_size + ball_size + kickable_margin
end if
command = closeIntercept(dist)
if command ≠ CMD_ILLEGAL then
  return command
end if
i = 1
repeat
  i = i + 1
  q̄^{t+i} = predictGlobPosAfterNrCycles(OBJECT_BALL, i)        // see Section 6.5
  n = predictNrCyclesToPoint(OBJECT_AGENT, q̄^{t+i})             // see Section 6.5
until n < i or i > InterceptMaxCycles
if i ≤ InterceptMaxCycles then
  return moveToPos(q̄^{t+i}, InterceptTurnAngle, InterceptDistanceBack, false)
else
  return CMD_ILLEGAL
end if
```

Algorithm 7.6: Pseudo-code implementation for intercepting the ball.

7.4.2 Dribbling

This skill enables an agent to dribble with the ball, i.e. to move with the ball while keeping it within a certain distance. This amounts to repeatedly kicking the ball at a certain speed into a desired direction and then intercepting it again. Two arguments, α and *type*, are supplied to this skill which respectively denote the global direction towards which the agent wants to dribble and the kind of dribble that must be performed. We distinguish three kinds of dribbling:

- `DRIBBLE_FAST`: a fast dribble action in which the agent kicks the ball relatively far ahead of him.
- `DRIBBLE_SLOW`: a slower dribble action in which the agent keeps the ball closer than on a fast dribble.
- `DRIBBLE_WITH_BALL`: a safe dribble action in which the agent keeps the ball very close to his body.

It is important to realize that this skill is only called when the ball is located within the agent's kickable range. This means that it is only responsible for the kicking part of the overall dribbling behavior, i.e. it only causes the ball to be kicked a certain distance ahead into the desired direction α. If the absolute angle between α and the agent's body direction is larger than *DribbleTurnAngle* (which currently has a value of 30 degrees) then the agent uses the `turnWithBallTo` skill to turn with the ball towards the global angle α. Otherwise, he uses the `kickTo` skill to kick the ball into the desired direction towards a point that lies a certain distance ahead depending on the type of dribble. After the kick, the ball will move out of the agent's kickable range and as a result the agent will try to intercept it using the `intercept` skill. The dribbling skill can then be called again once the agent has succeeded in intercepting the ball. This sequence of kicking and intercepting will repeat itself until the agent decides to perform another skill. Note that during the dribble the power of a kick depends on the distance that the ball should travel and on the speed that it should have when it reaches the target point. In our current implementation this speed equals 0.5 (=*DribbleKickEndSpeed*) for any type of dribble. Experiments have shown that lower end speed values cause the agent to intercept the ball before it reaches the target point which slows the dribble down significantly. A pseudo-code implementation for the dribbling skill is given in Algorithm 7.7.

```
dribble(α,type)

if type == DRIBBLE_WITH_BALL then
  dist = DribbleWithBallDist              // 2.0 in our current implementation
else if type == DRIBBLE_SLOW then
  dist = DribbleSlowDist                  // 3.0 in our current implementation
else if type = DRIBBLE_FAST then
  dist = DribbleFastDist                  // 7.0 in our current implementation
end if
if |ν((θ^t + φ^t) − α)| > DribbleTurnAngle then
  return turnWithBallTo(α, TurnWithBallAngle, TurnWithBallSpeed)
else
  return kickTo(q_t + π(dist, α), DribbleKickEndSpeed, KickMaxThr)     // q_t = current ball position
end if
```

Algorithm 7.7: Pseudo-code implementation for dribbling.

7.4.3 Passing the Ball Directly to a Teammate

This skill enables an agent to pass the ball directly to another player. It receives two arguments, o and *type*, which respectively denote the object (usually a teammate) to which the agent wants to pass the ball and the kind of pass (either normal or fast) that should be given. This skill uses the `kickTo` skill to pass the ball to the current location $\vec{q}_t$ of the object o with a certain desired end speed depending on the type of pass. A pseudo-code implementation for this skill is given in Algorithm 7.8.

```
directPass(o,type)

if type == PASS_NORMAL then
  return kickTo(q_t, PassEndSpeed, KickMaxThr)        // PassEndSpeed currently equals 1.4
else if type == PASS_FAST then
  return kickTo(q_t, FastPassEndSpeed, KickMaxThr) // FastPassEndSpeed currently equals 1.8
end if
```

Algorithm 7.8: Pseudo-code implementation for passing the ball directly to another player.

7.4.4 Giving a Leading Pass

This skill enables an agent to give a leading pass to another player. A leading pass is a pass into open space that 'leads' the receiver, i.e. instead of passing the ball directly to another player it is kicked just ahead of him. In this way, the receiver is able to intercept the ball while moving in a forward direction and this will speed up the attack. This skill receives two arguments, an object o (usually a teammate) and *dist*, which respectively denote the intended receiver of the leading pass and the 'leading distance' ahead of the receiver. It uses the `kickTo` skill to pass the ball to a point that lies *dist* in front of the current position $\vec{o}_t$ of o. Here 'in front of' means in positive x-direction, i.e. at a global angle of 0 degrees. In our current implementation *dist* is usually equal to the parameter *LeadingPassDist* which has a value of 1.5 meters. Note that the desired end speed for a leading pass is always equal to *PassEndSpeed* (currently 1.4) since the leading aspect of the pass might cause the receiver to miss the ball when its speed is higher. A pseudo-code implementation for this skill is given in Algorithm 7.9.

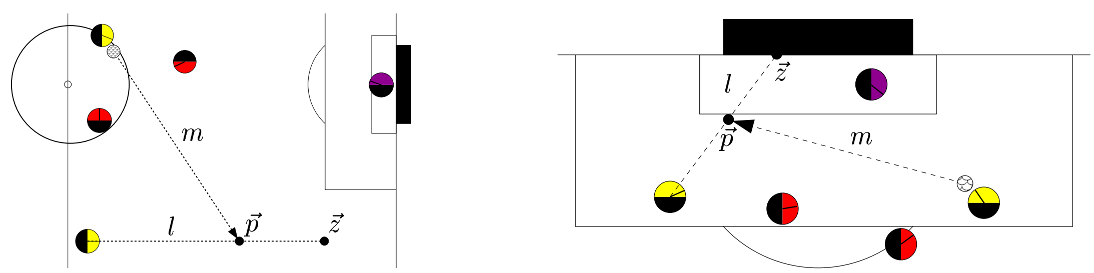
```
leadingPass(o,dist)

return kickTo(o_t + π(dist,0), PassEndSpeed, KickMaxThr)
```

Algorithm 7.9: Pseudo-code implementation for giving a leading pass.

7.4.5 Passing the Ball into the Depth (Through Pass)

This skill enables an agent to give a more advanced type of pass called a through pass. With a through pass the ball is not passed directly to another player or just ahead of him, but it is kicked into open space between the opponent defenders and the opponent goalkeeper in such a way that a teammate (usually an attacker) will be able to reach the ball before an opponent does. If a through pass is executed successfully it often causes a disorganization of the opponent's defense which will enable an attacker to get the ball close to the enemy goal. This skill takes an object o (usually a teammate) as an argument which denotes the intended receiver of the through pass. The position $\vec{p}$ on the field to which the ball should be kicked is determined by drawing a line l from the object's current position $\vec{o}_t$ to a given point $\vec{z}$ (also supplied as an argument) and by computing the safest trajectory for the ball to a point on this line. To this end, the ψ function is used to calculate the widest angle between opponents from the current ball position $\vec{q}_t$ to a point $\vec{p}$ on l. After this, the speed that the ball should have when it reaches the point $\vec{p}$ is determined based on the distance from the current ball position $\vec{q}_t$ to $\vec{p}$ and on the number of cycles n that o will need to reach $\vec{p}$. If it turns out that the required end speed falls outside the range [*MinPassEndSpeed*, *MaxPassEndSpeed*] it is set to the closest boundary of this range. The `kickTo` skill is then used to kick the ball to the desired point $\vec{p}$ at the required speed. Two example situations are shown in Figure 7.7. In Figure 7.7(a) the player with the ball cannot safely pass the ball directly to his teammate located at the bottom of the picture and decides to give a through pass. As a result, the ball moves between the opponent defenders to the other side of the field thereby disorganizing the opponent's defense while the team keeps possession of the ball. Figure 7.7(b) shows a situation in which the through pass leads to a scoring opportunity. A pseudo-code implementation for through passing is given in Algorithm 7.10.



(a) Disorganizing the opponent defenders. **(b)** Creating a scoring opportunity.

Figure 7.7: Two example situations for the `throughPass` skill. Red players are opponents.

```
throughPass(o, z⃗)

l = line segment that runs from o⃗_t (current position of o) to z⃗
α = ψ((o⃗_t − q⃗_t)_φ, (z⃗ − q⃗_t)_φ, (z⃗ − q⃗_t)_r)        // direction of widest angle between opponents
let m be the line that goes through q⃗_t into direction α and set p⃗ = intersection(m,l)
n = predictNrCyclesToPoint(o, p⃗)        // see Section 6.5
kick_speed = min(ball_speed_max, getFirstInGeometricSeries((p⃗ − q⃗_t)_r, ball_decay, n))
end_speed = kick_speed · ball_decay^n
if end_speed < MinPassEndSpeed then
  end_speed = MinPassEndSpeed
else if end_speed > MaxPassEndSpeed then
  end_speed = MaxPassEndSpeed
end if
return kickTo(p⃗, end_speed, KickMaxThr)
```

Algorithm 7.10: Pseudo-code implementation for through passing.

7.4.6 Outplaying an Opponent

This skill enables an agent to outplay an opponent. It is used, for example, when an attacker wants to get past an enemy defender. This is done by passing the ball to open space behind the defender in such a way that the attacker can beat the defender to the ball. Note that the attacker has an advantage in this situation, since he knows to which point he is passing the ball and is already turned in the right direction, whereas the defender is not. As result, the attacker has a headstart over the defender when trying to intercept the ball. Since a player can move faster without the ball, the main objective is to kick the ball as far as possible past the opponent while still being able to reach it before the opponent does. This skill receives two arguments, $\vec{q}$ and o, which respectively denote the point to which the agent wants to move with the ball and the object (usually an opponent) that the agent wants to outplay in doing so. First it is determined if it is possible to outplay the opponent o in the current situation. Let l be the line segment that runs from the agent's current position $\vec{p}_t$ to the given point $\vec{q}$. The best point to kick the ball to will be the furthest point from $\vec{p}_t$ on this line that the agent can reach before the opponent does. We use a simple geometric calculation to find the point $\vec{s}$ on l which has equal distance to the agent and to the opponent. Let $\vec{o}\,'$ be the perpendicular projection of the opponent's position $\vec{o}_t$ onto l and let d_1, d_2 and d_3 respectively denote the distance between $\vec{p}_t$ and $\vec{o}\,'$, the distance between $\vec{o}_t$ and $\vec{o}\,'$ and the distance between $\vec{o}\,'$ and $\vec{s}$ (see Figure 7.8). To determine $\vec{s}$, we need to compute the unknown value of d_3 using the values for d_1 and d_2 which can be derived from the world model. Since the distance from $\vec{p}_t$ to $\vec{s}$ will be equal to the distance from $\vec{o}_t$ to $\vec{s}$, the Pythagorean theorem guarantees that the following must hold:

$$d_1 + d_3 = \sqrt{d_2^2 + d_3^2} \quad \longleftrightarrow \quad d_3 = \frac{d_2^2 - d_1^2}{2d_1} \tag{7.22}$$

Using this value for d_3, we can compute the coordinates of the shooting point $\vec{s}$ as follows:

$$\vec{s} \;=\; \vec{p}_t + \pi(d_1 + d_3, (\vec{q} - \vec{p}_t)_\phi) \tag{7.23}$$

However, in some situations it is not likely that shooting the ball to this point will eventually result in outplaying the given opponent o on the way to $\vec{q}$. We therefore use the values for d_1, d_2 and d_3 to determine whether it is possible to outplay o in the current situation, and if so, what the best shooting point will be. The following situations are distinguished:

- $d_1 + d_3 \geq$ *OutplayMinDist*. Here *OutplayMinDist* is a parameter which has a value of 7.0 meters in our current implementation. If this condition holds, the opponent is located at a relatively large

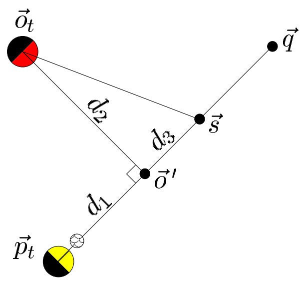


Figure 7.8: Determining the optimal shooting point $\vec{s}$ when outplaying an opponent (red player) on the way to $\vec{q}$. The values for d_1 and d_2 are known and can be used to compute d_3.

distance from the agent which makes an attempt to outplay him likely to be successful. First it is checked whether the agent's body is turned sufficiently towards the point $\vec{q}$. If this is not the case then the `turnWithBallTo` skill is used to turn with the ball in the right direction. Otherwise, the `kickTo` skill is used to kick the ball to a point on the line l where the agent will be able to intercept it first. Note that in general the agent will be able to reach the point $\vec{s}$ before the opponent despite the fact that both players need to travel the same distance to this point. This is because the agent has already turned his body more or less towards $\vec{q}$ (and thus towards $\vec{s}$), whereas the opponent probably has not. However, the actual point $\vec{z}$ to which the ball is kicked is chosen slightly closer to the agent than the point $\vec{s}$ defined by (7.23) in order to be absolutely sure that he can intercept the ball before the opponent does. This safety margin is represented by the parameter *OutplayBuffer* which has a value of 2.5 meters in our current implementation. The shooting point $\vec{z}$ then becomes:

$$\vec{z} \quad = \quad \vec{p}_t + \pi(d_1 + d_3 - \mathit{OutplayBuffer}, (\vec{q} - \vec{p}_t)_\phi) \tag{7.24}$$

For this skill the desired end speed when the ball reaches $\vec{z}$ equals *OutplayKickEndSpeed* (=0.5). Note that the value for this parameter cannot be chosen too low, since this will cause the agent to intercept the ball before it reaches the target point (see also Section 7.4.2).

- $d_1 + d_3 < \mathit{OutplayMinDist}$ and $d_1 < d_2$. If this condition holds, the opponent is located close to the agent which makes it difficult to outplay him. However, if the agent is already turned in the right direction (i.e. towards $\vec{q}$) then $d_1 < d_2$ implies that the distance between the opponent and the line l (denoting the agent's desired movement trajectory) is large enough for the agent to outplay this opponent when the ball is kicked hard in the direction of $\vec{q}$ (i.e. further ahead than $\vec{s}$). This is because the agent can start dashing after the ball immediately, whereas the opponent still has to turn in the right direction. As a result, the agent will have dashed past the opponent by the time the latter has turned correctly and this puts him in a good position to intercept the ball before the opponent. Therefore it is checked first whether the agent's body is sufficiently turned towards $\vec{q}$ and if this is not so then the `turnWithBallTo` skill is used to turn with the ball in the right direction. Otherwise, the `kickTo` skill is used to kick the ball past the opponent. In this case the point $\vec{z}$ to which the ball is kicked either lies *OutplayMaxDist* (=20) meters ahead of the agent's current position $\vec{p}_t$ into the direction of $\vec{q}$ or equals $\vec{q}$ when the distance to $\vec{q}$ is smaller than this value.
- In all other cases (i.e. $d_1 + d_3 < \mathit{OutplayMinDist}$ and $d_1 \geq d_2$) this skill returns an illegal command to indicate that it is not possible to outplay the opponent o on the way to the point $\vec{q}$.

Figure 7.9 shows three steps in the process of outplaying an opponent on the way to a point $\vec{q}$. In the first situation (I) the distance between the agent and the opponent is quite large. However, if the agent would

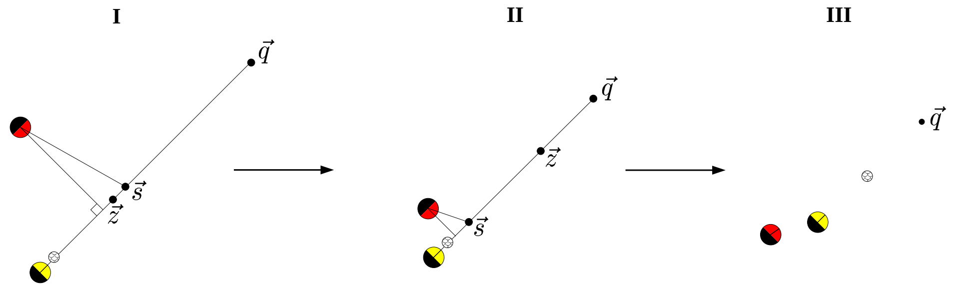


Figure 7.9: Three steps in the process of outplaying an opponent (red player) while moving with the ball to a position $\vec{q}$. Here $\vec{s}$ denotes the point along the desired trajectory which has equal distance to the agent and to the opponent and $\vec{z}$ is the point to which the ball is actually kicked.

kick the ball directly towards the point $\vec{q}$ then the opponent would be able to intercept it first. The agent therefore kicks the ball to the point $\vec{z}$ where he can reach it before the opponent can. After the kick, both the agent and the opponent start moving towards the ball which the agent reaches first. This is shown in the second situation (II). The opponent has now come close to the agent but is not orientated towards $\vec{q}$. The agent therefore decides to kick the ball hard to the point $\vec{z}$ and immediately starts to move after it. Since the opponent still has to turn in the right direction he will lose time and this enables the agent to dash past him. The result is depicted in the third situation (III) which shows that the opponent has been successfully outplayed. A pseudo-code implementation for this skill is given in Algorithm 7.11.

<u>`outplayOpponent`($\vec{q}, o$)</u>

l = line segment that runs from the agent's current position $\vec{p}_t$ to $\vec{q}$
$\vec{o}\,'$ = perpendicular projection of $\vec{o}_t$ (current position of o) onto l
$d_1 = (\vec{o}\,' - \vec{p}_t)_r$, $d_2 = (\vec{o}\,' - \vec{o}_t)_r$, $d_3 = (d_2^2 - d_1^2)/2d_1$
if $d_1 + d_3 \geq$ *OutplayMinDist* **then**
  $\vec{z} = \vec{p}_t + \pi(d_1 + d_3 - \mathit{OutplayBuffer}, (\vec{q} - \vec{p}_t)_\phi)$
else if $d_1 + d_3 <$ *OutplayMinDist* **and** $d_1 < d_2$ **then**
  $\vec{z} = \vec{p}_t + \pi(\min(\mathit{OutplayMaxDist}, (\vec{q} - \vec{p}_t)_r), (\vec{q} - \vec{p}_t)_\phi)$
else
  return `CMD_ILLEGAL`
end if
if $|(\vec{q} - \vec{p}_t)_\phi - (\theta^t + \phi^t)| >$ *OutplayTurnAngle* **then**
  return `turnWithBallTo`($(\vec{q} - \vec{p}_t)_\phi$, *TurnWithBallAngle*, *TurnWithBallSpeed*)
else
  return `kickTo`($\vec{z}$, *OutplayKickEndSpeed*, *KickMaxThr*)
end if

Algorithm 7.11: Pseudo-code implementation for outplaying an opponent.

7.4.7 Clearing the Ball

This skill enables an agent to clear the ball to a certain area on the field. It is useful, for example, when a defender cannot dribble or pass the ball to a teammate in a dangerous situation. Using this skill he can then kick the ball up the field away from the defensive zone. It is important to realize that this skill is only called when the agent has no alternative options in the current situation. Clearing the ball amounts to kicking it with maximum power into the widest angle between opponents in a certain area. The shooting direction is determined using the ψ function which returns the direction of the bisector of this widest angle. The area on the field from which this angle is selected depends on the type of clear which is supplied as an argument to this skill. We distinguish three types of clearing:

- `CLEAR_BALL_DEFENSIVE`: clearing the ball away from the defensive zone into a triangular area which is defined by the current ball position $\vec{q}_t$ and the center line on the field.
- `CLEAR_BALL_OFFENSIVE`: clearing the ball towards the offensive zone into a triangular area which is defind by the current ball position $\vec{q}_t$ and the line segment that coincides with the front line of the opponent's penalty area and extends to the left and right side lines.
- `CLEAR_BALL_GOAL`: clearing the ball into a triangular area in front of the opponent's goal which is defined by the current ball position $\vec{q}_t$ and the line segment that runs from the center of the opponent's goal to the center of the front line of the penalty area.

An example of the clearing area for each type of clear is shown in Figure 7.10. The area starting from the leftmost player corresponds to the `CLEAR_BALL_DEFENSIVE` type, whereas the areas drawn from the middle and rightmost players correspond to the `CLEAR_BALL_OFFENSIVE` and `CLEAR_BALL_GOAL` types respectively. A pseudo-code implementation for this skill is given in Algorithm 7.12.

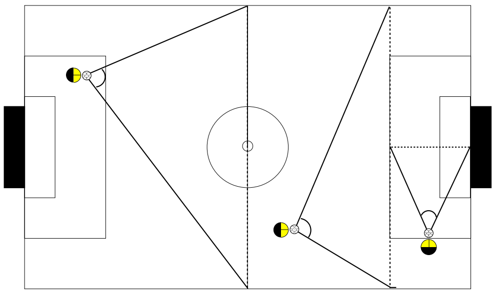


Figure 7.10: An example of the clearing area for each type of clear. From left to right the areas respectively correspond to the `CLEAR_BALL_DEFENSIVE`, `CLEAR_BALL_OFFENSIVE` and `CLEAR_BALL_GOAL` types.

<u>`clearBall`(*type*)</u>

if *type* == `CLEAR_BALL_DEFENSIVE` **then**
  $\vec{p}_1 = (0, -\texttt{PITCH\_WIDTH}/2)$, $\vec{p}_2 = (0, \texttt{PITCH\_WIDTH}/2)$
else if *type* == `CLEAR_BALL_OFFENSIVE` **then**
  $\vec{p}_1 = (\texttt{PENALTY\_X}, -\texttt{PITCH\_WIDTH}/2)$, $\vec{p}_2 = (\texttt{PENALTY\_X}, \texttt{PITCH\_WIDTH}/2)$
else if *type* == `CLEAR_BALL_GOAL` **and** $q_y^t > 0$ **then**
  $\vec{p}_1 = (\texttt{PENALTY\_X}, 0)$, $\vec{p}_2 = (\texttt{GOAL\_LINE\_X}, 0)$
else if *type* == `CLEAR_BALL_GOAL` **and** $q_y^t \leq 0$ **then**
  $\vec{p}_1 = (\texttt{GOAL\_LINE\_X}, 0)$, $\vec{p}_2 = (\texttt{PENALTY\_X}, 0)$
end if
$\alpha_{min} = (\vec{p}_1 - \vec{q}_t)_\phi$, $\alpha_{max} = (\vec{p}_2 - \vec{q}_t)_\phi$
$\alpha_{shoot} = \psi(\alpha_{min}, \alpha_{max}, \max((\vec{p}_1 - \vec{q}_t)_r, (\vec{p}_2 - \vec{q}_t)_r))$
return `kickTo`$(\vec{q}_t + \pi(1.0, \alpha_{shoot})$, `ball_speed_max`, *KickMaxThr*$)$

Algorithm 7.12: Pseudo-code implementation for clearing the ball.

7.4.8 Marking an Opponent

This skill enables an agent to mark an opponent, i.e. to guard him one-on-one with the purpose to minimize his usefulness for the opponent team. It can be used, for example, to block the path from the ball to an opponent or from an opponent to the goal. In this way, the agent can prevent this opponent from receiving a pass or from moving closer to the goal while also obstructing a possible shot. This skill amounts to calculating the desired marking position based on the given arguments and then moving to this position. It receives three arguments: an object o (usually an opponent) that the agent wants to mark, a distance d representing the desired distance between o and the marking position and a *type* indicator that denotes the type of marking that is required. We distinguish three types of marking:

- `MARK_BALL`: marking the opponent by standing at a distance d away from him on the line between him and the ball. In this case the marking position $\vec{z}$ is computed as follows:

$$\vec{z} \quad = \quad \bar{o}^{t+1} + \pi(d, (\bar{q}^{t+1} - \bar{o}^{t+1})_\phi) \tag{7.25}$$

  where $\bar{o}^{t+1}$ and $\bar{q}^{t+1}$ respectively denote the predicted positions of the opponent and the ball in the next cycle. This type of marking will make it difficult for the opponent to receive a pass.

- `MARK_GOAL`: marking the opponent by standing at a distance d away from him on the line between him and the center point $\vec{c}_g$ of the goal he attacks. The marking position $\vec{z}$ then becomes:

$$\vec{z} \quad = \quad \bar{o}^{t+1} + \pi(d, (\vec{c}_g - \bar{o}^{t+1})_\phi) \tag{7.26}$$

  This type of marking will make it difficult for the opponent to score a goal.

- `MARK_BISECTOR`: marking the opponent by standing at a distance d away from him on the bisector of the ball-opponent-goal angle. The marking position $\vec{z}$ is now computed as follows:

$$\vec{z} \quad = \quad \bar{o}^{t+1} + \pi(d, \mu((\bar{q}^{t+1} - \bar{o}^{t+1})_\phi, (\vec{c}_g - \bar{o}^{t+1})_\phi)) \tag{7.27}$$

  where the function μ computes the average of the angles which are supplied to it (see Section 7.1). This type of marking enables the agent to intercept both a direct and a leading pass to the opponent.

Figure 7.11 shows the marking positions for each of these types of marking in an example situation. The point $\vec{z}_1$ corresponds to the marking position for the `MARK_BALL` type, whereas the points $\vec{z}_2$ and $\vec{z}_3$ correspond to the marking positions for the `MARK_GOAL` and `MARK_BISECTOR` types respectively.

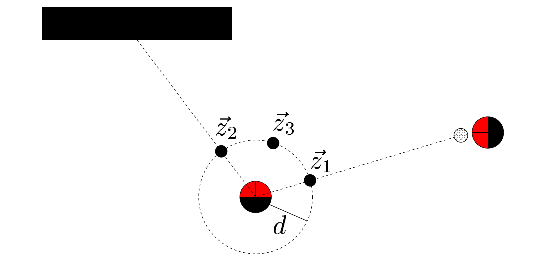


Figure 7.11: Three ways to mark an opponent. The marking positions $\vec{z}_1$, $\vec{z}_2$ and $\vec{z}_3$ respectively correspond to the `MARK_BALL`, `MARK_GOAL` and `MARK_BISECTOR` types of marking.

After determining the marking position $\vec{z}$, the agent uses the `moveToPos` skill to move to this position. Note that the decision whether to **turn** or **dash** in the current situation depends on the angle of $\vec{z}$

relative to the agent's body direction and on the distance to $\vec{z}$ if this point lies behind the agent. In this case, the `moveToPos` skill uses the threshold parameters *MarkTurnAngle* (=30) and *MarkDistanceBack* (=3) to make this decision. The values for these parameters are such that the condition which must hold for allowing a **dash** is fairly flexible. This is done because the marking position $\vec{z}$ will be different in consecutive cycles due to the fact that the opponent and the ball move around from each cycle to the next. As a result, the agent will be able to actually progress towards a point that lies close to $\vec{z}$ instead of constantly turning towards the newly calculated marking position in each cycle. A pseudo-code implementation for marking an opponent is given in Algorithm 7.13.

`markOpponent`(o, d, *type*)

if *type* == `MARK_BALL` **then**
  $\vec{z} = \vec{o}^{t+1} + \pi(d, (\vec{q}^{t+1} - \vec{o}^{t+1})_\phi)$
else if *type* ==`MARK_GOAL` **then**
  $\vec{z} = \vec{o}^{t+1} + \pi(d, (\vec{c}_g - \vec{o}^{t+1})_\phi)$
else if *type* == `MARK_BISECTOR` **then**
  $\vec{z} = \vec{o}^{t+1} + \pi(d, \mu((\vec{q}^{t+1} - \vec{o}^{t+1})_\phi, (\vec{c}_g - \vec{o}^{t+1})_\phi))$
end if
return `moveToPos`($\vec{z}$, *MarkTurnAngle*, *MarkDistanceBack*, *false*)

Algorithm 7.13: Pseudo-code implementation for marking an opponent.

7.4.9 Defending the Goal Line (Goaltending)

This skill enables an agent (usually the goalkeeper) to defend his own goal line. To this end, the agent moves to a position $\vec{z}$ along a line l which runs parallel to the goal line at a small distance d (supplied as an argument) from the goal. The guard point $\vec{z}$ to which the agent moves depends on the predicted position $\vec{q}^{t+1}$ of the ball in the next cycle and is chosen in anticipation of a future shot on goal. This means that the guard point is selected in such a way that it will be most difficult for the opponent team to pass the goalkeeper. To find this point we need to look at the angle that the ball makes with respect to the left and right goalposts and we need to determine which point on l covers this angle in the most optimal way. Let k and m be the line segments that run from the predicted ball position $\vec{q}^{t+1}$ to the left and right goalposts respectively and let $\vec{s}_1$ and $\vec{s}_2$ be the intersection points between k and l and between m and l. Furthermore, let d_1 and d_2 be the distances from $\vec{q}^{t+1}$ to $\vec{s}_1$ and from $\vec{q}^{t+1}$ to $\vec{s}_2$. The optimal guard point $\vec{z}$ on l can then be defined as the point on l for which the ratio between the distances e_1 and e_2 from $\vec{z}$ to $\vec{s}_1$ and from $\vec{z}$ to $\vec{s}_2$ is equal to the ratio between d_1 and d_2. In this way the distance that the goalkeeper must travel on l to intercept the ball is proportional to the distance that the ball must travel before it crosses l. Note that in our current implementation the goalkeeper always stays in front of the goal-mouth to avoid leaving an open goal when the ball is passed to an opponent in the center of the penalty area. The computed guard point is therefore adjusted if it lies too far to the side[6]. After computing the guard point $\vec{z}$, the goalkeeper needs to move to this point while keeping sight of the ball. If the distance between the current goalkeeper position $\vec{p}_t$ and the line l is larger than *DefendGoalLineMaxDist* (which currently has a value of 3.0 meters) the `moveToPos` skill is used to move directly towards $\vec{z}$. This can happen, for example, when the goalkeeper has moved forward from his line to intercept the ball and now has to move back to his line again. Note that the fourth argument supplied to the `moveToPos` skill equals *true* to indicate that the goalkeeper wants to turn his back to $\vec{z}$ in order to keep the ball in sight while moving. However, if the distance between $\vec{p}_t$ and l is less than *DefendGoalLineMaxDist* then the `moveToPosAlongLine` skill is used to move along l to the point $\vec{z}$. Recall from Section 7.3.7 that this skill receives an argument 'sign'

[6] Note that this also protects the goalkeeper from moving very far to the side if the ball lies behind the line l.

representing a prediction of the agent's future movement. This value is used to adjust the agent's body direction when necessary. In this case it can be expected that the goalkeeper will move along l in the same direction as the ball and 'sign' is therefore determined by looking at the ball velocity $\vec{w}_t$ in cycle t.

Figure 7.12 shows an example situation in which the goalkeeper (green player) is positioned at the optimal guard point on the line l. Note that the goalkeeper's body is aligned with l to enable him to move quickly along this line while keeping the number of turns to a minimum. However, the movement noise introduced by the *soccer server* will cause him to deviate from l as he moves. The body direction of the goalkeeper is therefore slightly adjusted in the `moveToPosAlongLine` skill as soon as this deviation becomes too large. This will enable him to move back towards l in subsequent cycles. Note that if the goalkeeper's body would be orientated more towards the center of the field it would also be harder for him to intercept a shot on goal since he would then need to turn to the right direction first. By aligning his body with l however, he is able to move immediately to the intersection point between l and the trajectory of the ball without turning. A pseudo-code implementation for this skill is given in Algorithm 7.14[7].

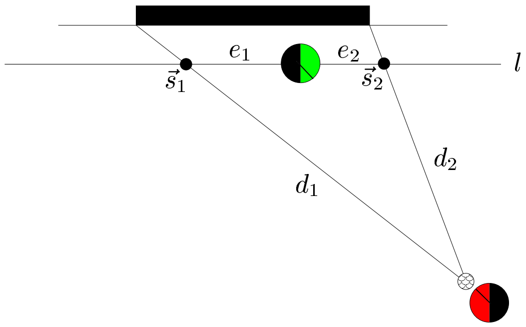


Figure 7.12: The optimal guard point on the line l in an example situation. It holds that $\frac{d_1}{d_2} = \frac{e_1}{e_2}$.

<u>`defendGoalLine`(d)</u>

l = line parallel to the goal line at distance d from the goal
k = line segment from $\bar{q}^{t+1}$ to left goalpost, m = line segment from $\bar{q}^{t+1}$ to right goalpost
$\vec{s}_1$ = intersection(k,l), $\vec{s}_2$ = intersection(m,l), $d_1 = (\bar{q}^{t+1} - \vec{s}_1)_r$, $d_2 = (\bar{q}^{t+1} - \vec{s}_2)_r$
$\vec{z} = \vec{s}_1 + \pi(\frac{d_1 \cdot (\vec{s}_2 - \vec{s}_1)_r}{d_1 + d_2}, (\vec{s}_2 - \vec{s}_1)_\phi)$ // the desired guard point
if $|z_y| >$ `GOAL_WIDTH`/2 **then**
  $z_y = sign(z_y) \cdot$ `GOAL_WIDTH`/2 // $sign(x) = 1$ if $x \geq 0$ and -1 otherwise
end if
if $(p_x^t - z_x) >$ *DefendGoalLineMaxDist* **then**
  return `moveToPos`$(\vec{z}$, *DefendGoalLineTurnAngle*, *DefendGoalLineDistanceBack*, *true*)
else
  $\alpha = sign(q_y^t - z_y) \cdot 90$ // desired orientation in the point $\vec{z}$ (keeping the ball in sight)
  sign = $sign(w_y^t)$ // $\vec{w}_t$ = ball velocity in cycle t
  return `moveToPosAlongLine`$(\vec{z}$, α, *DefendGoalLineDeviationDistance*, sign,
    DefendGoalLineTurnThr, *DefendGoalLineCorrectionAngle*)
end if

Algorithm 7.14: Pseudo-code implementation for defending the goal line.

[7] *DefendGoalLineTurnAngle*, *DefendGoalLineDistanceBack*, *DefendGoalLineDeviationDistance*, *DefendGoalLineTurnThr* and *DefendGoalLineCorrectionAngle* have respective values of 15.0, 10.0, 0.75, 4.0 and 10.0 in our current implementation.

Chapter 8

Agent Scoring Policy

One of the main objectives in a soccer game is to score goals. It is therefore important for a robotic soccer agent to have a clear policy about whether he should attempt to score in a given situation, and if so, which point in the goal he should aim for. In this chapter we describe the implementation of the scoring policy that was used by the *UvA Trilearn* agents during the *RoboCup-2001* world championship. This scoring policy enables an agent to determine the best target point in the goal, together with an associated probability of scoring when the ball is shot to this point. It is partly based on an approximate method that we have developed for learning the relevant statistics of the ball motion which can be regarded as a geometrically constrained continuous-time Markov process. This method has led to the publication of a paper titled *'Towards an Optimal Scoring Policy for Simulated Soccer Agents'* [49] which has been accepted at the 7th International Conference on Intelligent Autonomous Systems (IAS-7) and which serves as the basis of this chapter (see also [22]). As such, we consider this chapter to be one of the main contributions of this thesis. It is organized as follows. In Section 8.1 we provide a general introduction to the scoring problem and show that it can be decomposed into two independent subproblems. A solution to each of these subproblems is presented in Sections 8.2 and 8.3. In Section 8.4 we then describe how these solutions can be combined to determine the best scoring point. Several remarks about the implementation of our scoring policy are included in Section 8.5 which also presents results concerning its performance. Finally, Section 8.6 contains an overall conclusion and briefly discusses possible extensions.

8.1 The Optimal Scoring Problem

An important decision for a robotic soccer agent is which action to choose when he has control of the ball. Possible options could be, for example, passing the ball to a teammate, dribbling with the ball, shooting the ball at the goal or clearing the ball up the field. Especially the decision whether to shoot at the goal or not can be a crucial one since scoring goals is one of the main objectives in a soccer game. It is therefore important for a robotic soccer agent to have a clear policy about whether he should attempt to score in a given situation, and if so, which point in the goal he should aim for. Ideally, the decision whether to shoot to the goal should be based on the probability of scoring given the current state of the environment. To achieve this, one will need to solve a problem which will be referred to as the *optimal scoring problem* and which can be stated as follows: *find the point in the goal where the probability of scoring is the highest when the ball is shot to this point in a given situation.* It can be expected that the probability of scoring will at least depend both on the position and angle of the ball with respect to the goal and on the position of the goalkeeper relative to the ball. However, solving the *optimal scoring problem* is not straightforward. The reason for this is that the total number of possible situations is extremely large[1] and that different variables can be decisive for different situations. Furthermore, the problem depends on many uncertain factors. For example, the noise in the ball movement can never be exactly predicted and will be different for different distances that the ball travels. To find the optimal scoring point by iterating over all possible goal points one thus has to take different functions into account, since the distance from the shooting position to the scoring point will be different for each point in the goal. On top of this, the behavior of the opponent goalkeeper cannot be easily predicted but is an important factor for solving the problem. As a result, no simple analytical solution to the problem exists and one has to look for different methods.

A key observation for solving *the optimal scoring problem* is that it can be decomposed into two independent subproblems:

1. Determining the probability that the ball will enter the goal when shot to a specific point in the goal from a given position.
2. Determining the probability of passing the goalkeeper in a given situation.

To solve the *optimal scoring problem* we thus have to find solutions to these two subproblems and combine them. Since the subproblems are independent, the probability of scoring when shooting at a certain point in the goal is equal to the *product* of the two probabilities. In the remainder of this chapter we will describe a statistical framework for computing these probabilities. Before we do this however, we need to mention a number of simplifying assumptions that have been made in our solution to the scoring problem.

Firstly, we assume that the ball is always shot to the goal with maximum power giving it an initial velocity of 2.7 (meters per simulation cycle) in the current version of the *soccer server* ($7.x$). When the ball is shot with less power its velocity will obviously be lower and as a result the movement noise per cycle will be less. In this case however, the ball will need more cycles to reach the target point and the movement noise will thus be added more often. The interaction between these factors slightly complicates the problem and since shooting to the goal with maximum power is the common case in practice, we have decided to make this assumption. Secondly, we have chosen to neglect the possibility that other players besides the goalkeeper are blocking the path to the goal. The reason for this is that it is much easier for the goalkeeper to intercept the ball than for a regular field player due to the fact that the goalkeeper is allowed to catch the ball. Furthermore, players from different teams do not intercept the ball equally well. Finding a correct interpretation of the angle between a goalkeeper and a field player when it comes to determining a safe trajectory for the ball is thus not straightforward. It is clear however, that the goalkeeper's superior interceptive capabilities make passing him the main objective and we have therefore chosen to neglect regular field players in our solution to the scoring problem. Finally, we also assume in our experiments that the ball is always shot from a distance smaller than 32 meters from the target point in the goal. This amounts to approximately one-third of the total field length (`pitch_length`=105 meters). Since the distance that the ball will travel when it is shot with maximum power equals about 45 meters (neglecting movement noise; see Section 3.4.1), it will then never be the case that the ball comes to a halt before it has reached the goal. Note that this assumption will yield a good approximation since it is almost impossible to score from larger distances anyway due to the fact that the goalkeeper usually has enough time in these cases to intercept the ball before it reaches the goal line. Although it is fairly straightforward to relax these assumptions and extend the method appropriately, we have chosen not to do this to avoid having to concentrate on details for situations which rarely occur.

[1] With the current server parameter values for `pitch_length` (=105m) and `pitch_width` (=68m) and with the current precision of visual observations (one decimal for coordinates and rounded to the nearest integer for angles), the perceived state space consists of more than 10^{226} states when only positions and orientations of body and neck are considered: each of the 22 players and the ball can be in any of 680×1050 positions and each player can have any of 360 body orientations and 180 possible neck angles. The state space becomes even larger when velocities and physical parameters are also considered.

8.2 The Probability that the Ball Enters the Goal

If there were no noise in the movement of the ball, it would always enter the goal when shot to a point inside the goal from a given position. However, due to the limited goal width and the noise introduced by the *soccer server*, the ball may miss the goal on an attempt to score. In this section we will show how one can determine the probability that the ball will end up *somewhere* inside the goal when shot at a *specific* goal point. To this end we first need to compute the deviation of the ball from the aiming point. This deviation is caused by the noise which is added to the ball velocity vector in each simulation cycle. Recall from Section 3.3 that the ball velocity (v_x^{t+1}, v_y^{t+1}) in cycle $t+1$ is equal to `ball_decay`$\cdot(v_x^t, v_y^t) + (\tilde{r}_1, \tilde{r}_2)$ where $\tilde{r}_1$ and $\tilde{r}_2$ are random numbers uniformly distributed over the interval $[-\mathrm{rmax}, \mathrm{rmax}]$ with rmax $=$ `ball_rand`$\cdot\|(v_x^t, v_y^t)\|$. Here `ball_decay` and `ball_rand` are server parameters which have respective values of 0.94 and 0.05. Note that the ball motion can be regarded as a *diffusion* process since the position of the ball in each time step diffuses over time. We can treat it as a Markov process because the future development is completely determined by the present state and is independent of the way in which the present state has developed, i.e. the ball velocity in cycle $t+1$ depends only on the current ball velocity and makes no reference to the past. Since the statistical properties of the process will be different for each time step, the process is called *non-stationary*[2].

An interesting aspect of the *soccer server* is that, although the underlying noise statistics for ball motion are known, finding an analytical solution of the corresponding diffusion process is difficult for two reasons:

- The motion noise which is added by the server in each cycle is by construction non-white due to the fact that this noise depends on the speed of the ball in the previous cycle as shown above.
- The process is geometrically constrained since the ball must end up inside the goal.

This makes an analytical computation of the process statistics nontrivial. When we treat ball motion as a continuous-time Markov process, computing the exact statistics for each time step would require the solution of a corresponding Fokker-Planck equation for diffusion with drift [30, 66]. The non-white noise complicates this however. Furthermore, this will lead to a solution which is not generic but dependent on the current server parameter values. To avoid both problems, we propose to *learn* the ball motion statistics from experiments and to calculate the required probabilities from these statistics. This gives an approximate solution to the problem of computing the statistical properties of a geometrically constrained continuous-time Markov process and we believe that this relatively simple alternative, which also allows for an easy adaptation when the server noise parameters change, can be useful in other applications as well (e.g. Brownian motion problems[3] [66]).

Our solution thus amounts to estimating the cumulative noise directly from experiments. To this end, we computed the deviation of the ball perpendicular to the shooting direction as a function of the distance that the ball had traveled. This function was learned by repeating an experiment in which the ball was placed at even distances between 0 and 32 meters in front of the center of the goal (zero y-coordinate) with a player directly behind it. The player then shot the ball 1,000 times from each distance with maximum power and perpendicularly to the goal line (i.e. towards the center of the goal). An example of this setup is depicted in Figure 8.1. For each instance we recorded the y-coordinate of the point on the goal line where the ball entered the goal. We then used these values to compute the sample standard deviation σ of the ball, which for n zero-mean points x_i is equal to $\sigma = \sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2}$. As expected, we saw that the

[2] A stochastic process $\mathbf{x}(t)$ is called *stationary* if its statistical properties are invariant to a shift of the origin. This means that the processes $\mathbf{x}(t)$ and $\mathbf{x}(t+c)$ have the same statistics for any c.

[3] The term *Brownian motion* is used to describe the movement of a particle in a liquid, subjected to collisions and other forces.

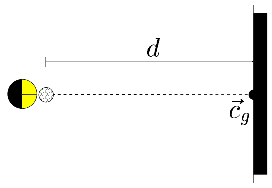


Figure 8.1: Experimental setup for learning the cumulative noise in the ball motion as a function of the traveled distance. The ball lies in front of the center of the goal at an even distance d (between 0 and 32 meters) from the goal line. The agent stands directly behind the ball and shoots it with maximum power to the point $\vec{c}_g$. The motion noise will cause the ball to deviate from the desired trajectory (dotted line).

cumulative noise was different for each distance d. We empirically found that the standard deviation σ of the ball perpendicular to the shooting direction was a monotone increasing function of the distance. To a good approximation this function could be represented by the following formula:

$$\sigma(d) \quad = \quad -1.88 \cdot \ln(1 - d/45) \tag{8.1}$$

where 'ln' denotes the natural logarithm. Note that this formula can only be used for distance values d between 0 and 32. No assumptions can be made for distances outside this interval, since no data were collected for these situations. Figure 8.2 shows a plot of the function together with the recorded deviation values. A qualitative explanation of this result is as follows. In each cycle the ball will have a remaining speed which is equal to 94% of its speed in the previous cycle. As the speed of the ball decreases, more cycles will thus be needed for it to travel the same distance. Since noise is added in each cycle, this means that the noise will be added to the ball motion more often. Although the added noise in each cycle decreases with decreasing speed, the results indicate that the cumulative noise for traveling a particular distance increases monotonically. The surprisingly simple formula of σ as a function of d implies that the standard deviation of the process increases linearly with time. This can be shown by solving the differential equation of the forward motion of the ball (ignoring noise) and expressing time as a function of the distance. The result contrasts with most Brownian motion problems for which $\sigma(t) = O(\sqrt{t})$ [66]. We expect that this difference can be mainly attributed to the non-white motion noise.

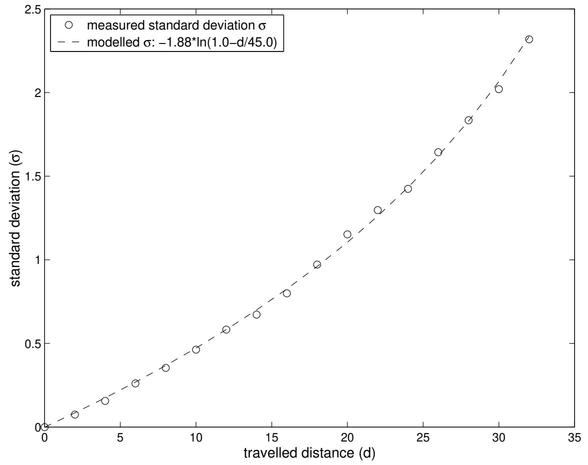


Figure 8.2: Standard deviation of the ball as a function of the traveled distance.

The next step is to compute the distribution of the ball when it reaches the goal line. For this we use a fundamental result from probability theory called the *Central Limit Theorem*. This theorem states that under certain conditions the distribution of the sum of N random variables x_i will be Gaussian as N goes to infinity [30]. Several versions of the Central Limit Theorem exist which have been proved under different conditions. In its most common form (due to Lindeberg [58]) the theorem has been shown to hold for sequences of mutually independent random variables with a common distribution. It is clear that this version does not apply to our current situation, since the motion noise in cycle t depends on the speed of the ball in cycle $t-1$. The required condition of independence is thus not satisfied in our case. It must be noted however, that the dependency is represented by the server parameter `ball_rand` which currently has a value of 0.05. This low value will cause only a slight dependence and we can therefore expect the independence condition to approximately hold. Furthermore, more general versions of the Central Limit Theorem exist (see [31]) which do not require a common distribution and even weaken the assumption of independence by allowing the x_i to be dependent as long as they are not 'too' dependent. Since the latter seems to be the case for the current problem, we can use this result to find a model for the distribution of the ball along the goal line. We note that the deviation of the ball is caused by noise which is added in each cycle and under the present conditions we can expect from the Central Limit Theorem that, whatever the distribution of this noise is, the *cumulative* noise will be approximately Gaussian. Moreover, this Gaussian g must have a zero mean and a standard deviation $\sigma = \sigma(d)$ from (8.1). This gives the following model:

$$g(y;\sigma) \quad = \quad \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{y^2}{2\sigma^2}\right) \tag{8.2}$$

Using this model, we can compute the probability that the ball will end up inside the goal when shot from an arbitrary position on the field perpendicularly to the goal line. This probability equals the area that lies under the respective Gaussian density function in between the two goalposts as is shown in Figure 8.3(a). When the y-coordinates of the left and right goalposts are respectively denoted by y_l and y_r with $y_l < y_r$, this can be computed as follows:

$$\mathrm{P(goal)} \quad = \quad \int_{y_l}^{y_r} g(y;\sigma)\,dy = \int_{-\infty}^{y_r} g(y;\sigma)\,dy - \int_{-\infty}^{y_l} g(y;\sigma)\,dy = G(y_r;\sigma) - G(y_l;\sigma) \tag{8.3}$$

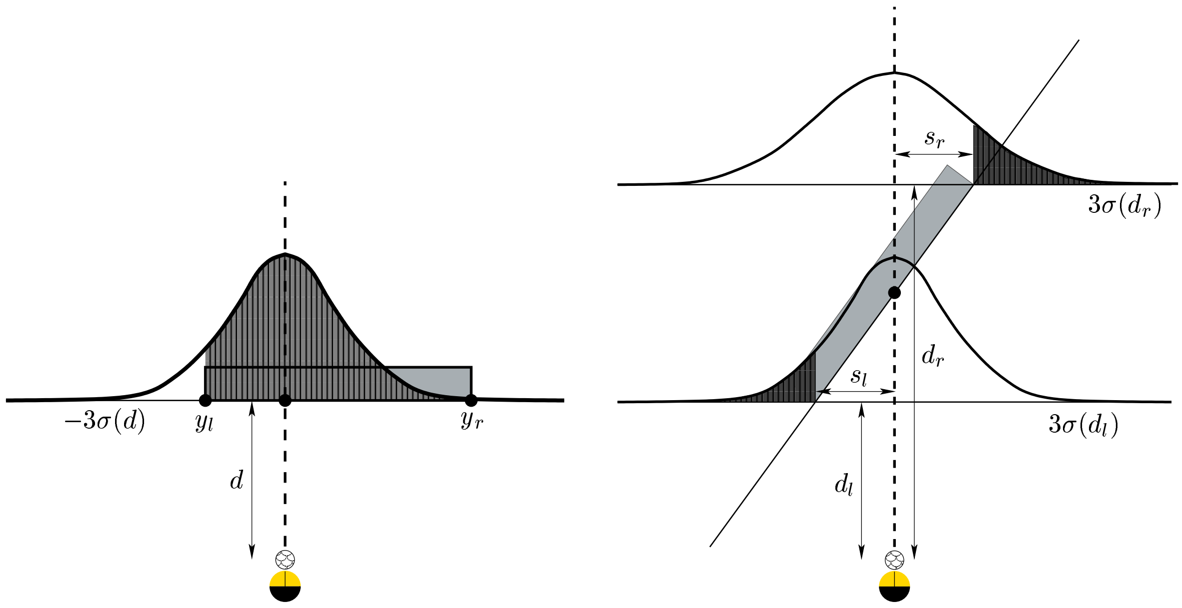
where $G(y;\sigma)$ denotes the cumulative distribution function of the Gaussian $g(y;\sigma)$. Note that $G(y;\sigma)$ can be implemented using the *erf* function which is defined in most mathematical libraries[4].

Finally, we have to compute the probability that the ball enters the goal when shot at an angle to the goal line (see Figure 8.3(b)). This case is somewhat more involved than the previous one due to the fact that the noise can cause the ball to travel different distances before it reaches the goal. Since different traveled distances imply different deviations according to (8.1), the ball distribution along the goal line is no longer Gaussian and this makes an exact calculation of the total probability difficult. A detailed analysis would involve bounding the corresponding diffusion process appropriately and computing the statistics for this bounded process which is a formidable task. However, the key observation is that we want to compute probability *masses* and that for equal masses the particular shape of the distribution that produces these masses is irrelevant. This observation directly motivates our solution to the problem: instead of computing the distribution of the ball along the goal line analytically (by solving the constrained diffusion process equations) and then integrating to find its probability mass between the two goalposts, we compute the probability mass from the identity

$$\mathrm{P(goal)} \quad = \quad 1 - \mathrm{P(not\ goal)} \tag{8.4}$$

where P(not goal) denotes the probability that the ball misses the goal, either going out from the left or the right goalpost. This probability mass is easier to compute, to a good approximation, from the tails

[4]Including the *math.h* library in *C* and *C++*.



(a) Shooting perpendicularly to the goal line. **(b)** Shooting at an angle to the goal line.

Figure 8.3: Two situations of shooting at the goal (light gray) together with the associated distributions.

of the Gaussian distributions corresponding to the two goalposts. This is shown in Figure 8.3(b): when the ball reaches the left goalpost it has *effectively* traveled distance d_l and its corresponding distribution perpendicular to the shooting line is Gaussian with deviation $\sigma(d_l)$ from (8.1). The probability that the ball misses the goal going out from the left goalpost is approximately[5] equal to the shaded area on the left in Figure 8.3(b), i.e.

$$\mathrm{P}(\text{out from left}) \approx \int_{-\infty}^{-s_l} g(y; \sigma(d_l))\, dy \tag{8.5}$$

where the integration runs up to $-s_l$ which denotes the (negative) shortest distance from the left goalpost to the shooting line. The situation that the ball misses the goal going out from the right post is analogous. The only difference is that the ball will have to travel a larger distance in this case. As a result, its deviation will be larger and the corresponding Gaussian will be flatter as can be seen in Figure 8.3(b). The respective probability is approximately equal to the shaded area on the right, i.e.

$$\mathrm{P}(\text{out from right}) \approx \int_{s_r}^{\infty} g(y; \sigma(d_r))\, dy \tag{8.6}$$

where the integration now runs from s_r, the shortest distance from the right goalpost to the shooting line, and where the Gaussian has a standard deviation $\sigma(d_r)$ which is obtained from (8.1) for a traveled distance d_r. Based on (8.4), the probability that the ball ends up inside the goal can now be written as

$$\mathrm{P}(\text{goal}) = 1 - \mathrm{P}(\text{out from left}) - \mathrm{P}(\text{out from right}) \tag{8.7}$$

which can be directly computed using Equations 8.5 and 8.6.

[5] There is a small probability that the ball ends up to the right of the left goalpost after traveling an 'illegal' trajectory outside the field. The ball thus actually went out from the left in this case but we neglect this probability in (8.5).

8.3 The Probability of Passing the Goalkeeper

Intercepting a moving ball is much easier for the goalkeeper than for a regular field player. The reason for this is that the goalkeeper is allowed to catch the ball, whereas a field player can only kick it. Since the catchable distance for a goalkeeper is larger than the kickable distance for a field player, this means that a field player must come closer to the ball in order to intercept it than the goalkeeper does. On an attempt to score a goal, these superior interceptive capabilities make passing the opponent goalkeeper one of the main objectives. In this section we present an approach for estimating the probability of passing the goalkeeper in a given situation. To be exact, the second subproblem in the optimal scoring task can be stated as follows: ***given a shooting point in the goal, determine the probability that the goalkeeper intercepts the ball before it reaches the goal line.*** We propose an empirical method for learning this probability from examples of successful and unsuccessful scoring attempts. Clearly, the problem depends heavily on the behavior of the opponent goalkeeper and unless a provably optimal goalkeeper behavior has been implemented (which is currently not the case) the experiments have to be based on existing goalkeeper implementations. In our experiments we have used the goalkeeper of *RoboCup-2000* winner *FC Portugal 2000*, since it appeared to be one of the best available goalkeepers.

To cast the problem into a proper mathematical framework, we note that ball interception by the goalkeeper on a scoring attempt can be regarded as a two-class classification problem: given the shooting point in the goal together with the positions of the goalkeeper and the ball (input feature vector), predict which class (intercepting or not) is most probable. Moreover, we are interested in the *posterior* probability associated with the prediction of each class. Formalizing the problem in this way allows for the direct application of various methods from the field of statistical pattern recognition [82]. To collect a training set, we performed an experiment in which a player repeatedly shot the ball from a fixed position straight to the goal, while the goalkeeper was placed randomly at different positions relative to the ball. The setup for this experiment is shown more precisely in Figure 8.4. A data set was formed by recording 10,000 situations, together with a boolean indicating whether the goalkeeper had intercepted the ball or not. An analysis of the resulting data revealed that the relevant features for classification were the following[6]:

- The absolute angle a between the goalkeeper and the shooting point as seen by the striker.
- The distance d between the ball and the goalkeeper.

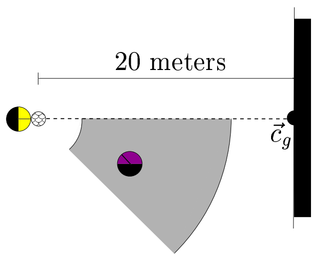
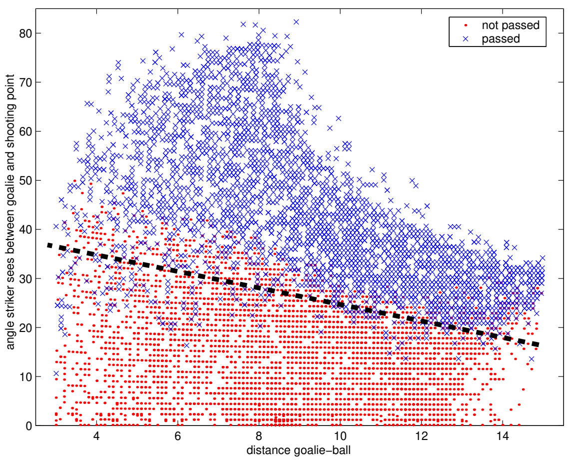


Figure 8.4: Experimental setup for learning the probability of passing the goalkeeper in a given situation. The ball lies in front of the center of the goal at a distance of 20 meters from the goal line. The striker (yellow player) stands directly behind the ball and shoots it with maximum power towards the point $\vec{c}_g$. The goalkeeper (purple player) is placed at a random position inside the shaded area, i.e. at an angle between 0 and 45 degrees and a distance between 3 and 15 meters from the ball.

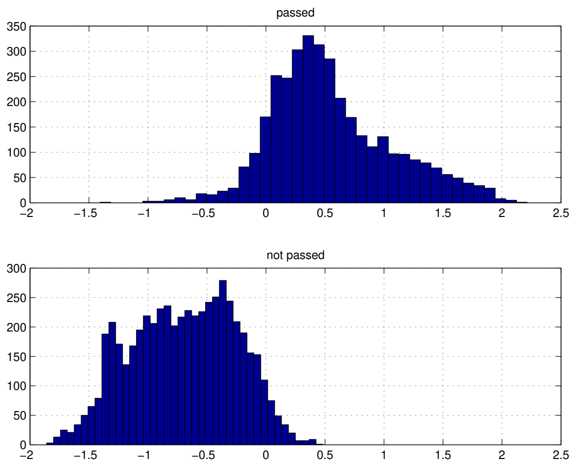
[6]Principled methods for automatic feature extraction are described in [82].

These two values form a two-dimensional feature vector on which the classification has been based. The recorded data set is depicted in Figure 8.5(a) which shows that there is an almost linear discriminant function between the two classes. We determined this discriminant function via linear regression on the boolean class indicator. This procedure is known to give the optimal *Fisher's Linear Discriminant* which has the property that it maximizes the ratio of the *between-group variance* and the *within-group variance* for the two classes. Details and definitions for this linear discrimination method can be found in [82]. The resulting discriminant function is characterized by the equation

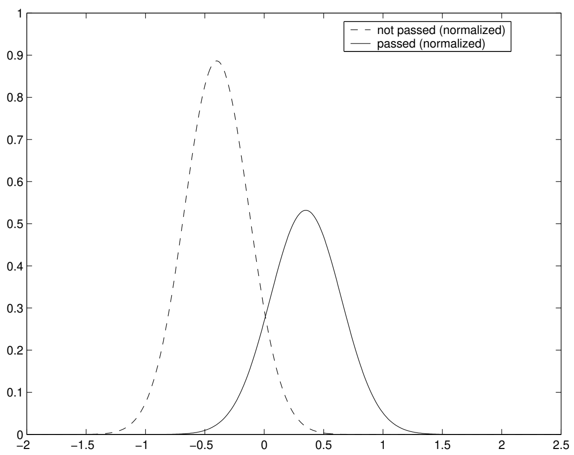
$$\begin{aligned} u &= (a - 26.1) * 0.043 + (d - 9.0) * 0.09 - 0.2 \\ &= 0.043 * a + 0.09 * d - 2.1323 \end{aligned} \tag{8.8}$$



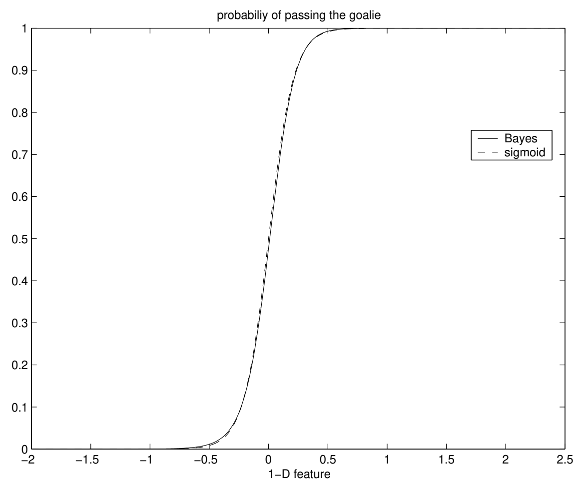
(a) Data set and discriminant function.



(b) 1-D class histograms.



(c) Gaussian approximations near discriminant.



(d) Estimated posterior probability of non-interception.

Figure 8.5: Data set for the goalkeeper interception experiment together with derived statistics.

for distance values d between 3 and 15. This can be interpreted as follows: for a new angle-distance pair (a, d), the sample mean $(26.1, 9.0)$ is subtracted from it after which the inner product (projection) with the vector $(0.043, 0.09)$ is computed. The resulting vector stands perpendicular to the discriminant boundary which is shifted appropriately by the offset -0.2. The pairs (a, d) for which Equation 8.8 equals zero form the boundary between the two classes. This is plotted by a dotted line in Figure 8.5(a).

Projecting all the (a_i, d_i) pairs perpendicularly to the discriminant line via Equation 8.8 gives a set of one-dimensional points u_i that, to a good approximation, describe the two classes. The histogram class distributions of these points are plotted in Figure 8.5(b). The upper histogram in this figure corresponds to the situations in which the goalkeeper did not succeed in intercepting the ball and the lower one to the situations in which he did intercept it. Instead of trying to model these two distributions parametrically, we note that the relevant range for classification is only where the two histograms *overlap*, i.e. the interval $[-0.5, 0.5]$. It is easy to see that the posterior probability of non-interception will be zero for approximately $u \leq -0.5$, one for $u > 0.5$ and will increase smoothly from zero to one in the interval in between. The posterior probability can thus be represented by a sigmoid. A principled way to find this sigmoid would be to optimize the unknown parameter with respect to the likelihood using a procedure known as *logistic discrimination* [82]. This amounts to fitting a posterior sigmoid with maximum likelihood directly from the available data. However, the low-dimensionality of the problem allows us to propose the following simpler solution. In the region where the class distributions for interception and non-interception overlap, we fit a univariate Gaussian function on each class as shown in Figure 8.5(c). For each class C this gives us a Gaussian model for the class-conditional density function $\mathrm{P}(u|C)$. With this model we can easily compute the posterior probability $\mathrm{P}(C|u)$ for a class C using the Bayes rule

$$\mathrm{P}(C|u) = \frac{\mathrm{P}(u|C)\,\mathrm{P}(C)}{\mathrm{P}(u|C)\,\mathrm{P}(C) + \mathrm{P}(u|\bar{C})\,\mathrm{P}(\bar{C})} \tag{8.9}$$

which is a sigmoid-like function. Since this is a simple two-class classification problem, $\bar{C}$ refers to the 'other' class in this case, while the prior probability $\mathrm{P}(C)$ for a class C is computed as the proportion of points u_i in the data set which belong to C. In Figure 8.5(d) we have plotted the posterior probability for the non-interception class as given by the Bayes rule, together with the sigmoid approximation

$$\mathrm{P}(\text{pass goalkeeper} \mid u) = \frac{1}{1 + \exp(-9.5u)} \tag{8.10}$$

which allows for an easy implementation.

8.4 Determining the Best Scoring Point

Having computed the probability that the ball will end up inside the goal (Equation 8.7) and the probability that the goalkeeper will not intercept it (Equation 8.10), the assumption of independence gives the total probability of scoring in a given situation as the product of these two values. In order to determine the best scoring point in the goal, we discretize the goal interval [−`goal_width`/2 , `goal_width`/2][7] and compute the total probability that the ball will end up in each discretized bin. This total probability is a bell-shaped function which represents the probability that the ball will enter the goal and which has a valley around the position of the goalkeeper. The best scoring point is determined by the global maximum of this curve. Note that the curve will always have two local maxima which correspond to the left and right starting point of the valley. These maxima can be located by using a simple *hill-climbing* search algorithm [84]: starting from the position of the goalkeeper (lowest point in the valley) we move to the

[7]The parameter `goal_width` has a value of 14.02 in the current version of the *soccer server*.

left and to the right along the curve as long as the function value increases. Of the two peaks which are found in this way, the highest one denotes the global maximum and is selected as the best scoring point.

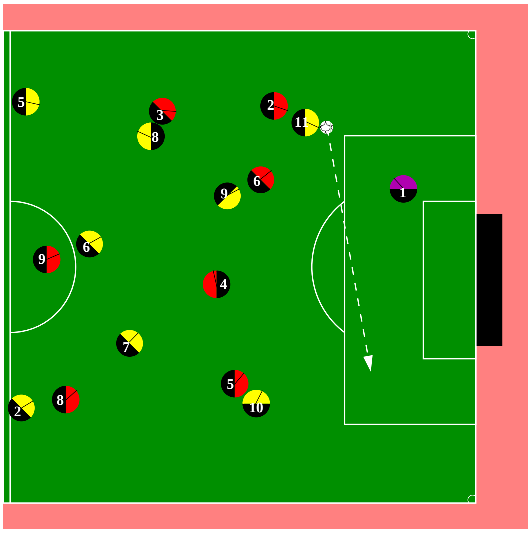
8.5 Implementation and Results

The scoring policy which has been described in this chapter has been implemented with a small number of modifications to improve the overall quality of the result in various situations. These modifications are needed because several aspects of our current solution to the *optimal scoring problem* are too specific. Especially our solution to the second subproblem (probability of passing the goalkeeper) is based on a number of assumptions which are not general. For example, the experiment described in Section 8.3 (see Figure 8.4) for learning the single probability of passing the goalkeeper in a given situation is always performed with the striker at a fixed distance of 20 meters from the center of the goal. As a result, the value returned by Equation 8.10 actually represents the probability that the goalkeeper will not intercept the ball before it has traveled a distance of 20 meters. However, using this definition in situations where the striker stands close to the goal can give inaccurate results since the ball might have already crossed the goal line before it will be intercepted by the goalkeeper. When the distance between the striker and the scoring point is less than 20 meters, the returned probability for passing the goalkeeper is thus too low. Furthermore, the probability of passing the goalkeeper has been solely based on the goalkeeper of *RoboCup-2000* winner *FC Portugal*. As a result, unnecessary risks might be taken with other goalkeepers which are often not so good. To compensate for these factors we made the following adjustments:

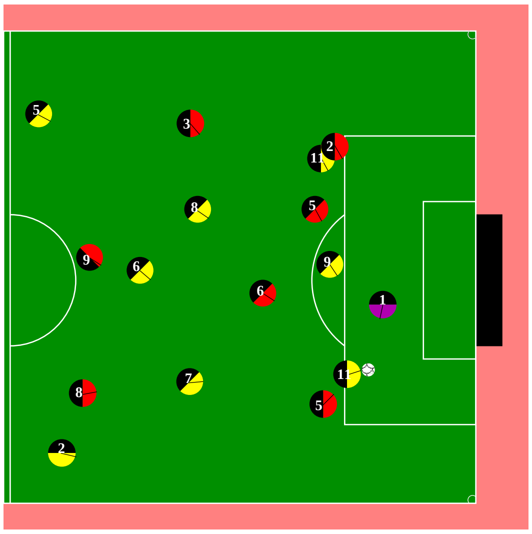
- If the ball is located within a distance of 20 meters from the scoring point, we first determine if it is theoretically possible for the goalkeeper to intercept it. This is done by computing the intermediate positions of the ball in each cycle before it reaches the goal line and by checking for each of these positions if in the optimal case the goalkeeper can reach it in time. To this end, the distance between the goalkeeper and the calculated ball position is compared to the maximum distance that the goalkeeper can cover in the given number of cycles. If it turns out that the goalkeeper will never be able to intercept the ball before it reaches the goal line, the returned probability of passing the goalkeeper is adjusted to 1.0. In all other cases this probability remains unaltered.
- To compensate for lower-quality goalkeepers and for the fact that the returned probability of passing the goalkeeper is too low if the distance to the scoring point is less than 20 meters, we only consider scoring points for which the respective single probability that the ball will enter the goal is larger than a specified threshold. This threshold is represented by the parameter *EnterGoalThr* which has a value of 0.7 in our current implementation. This ensures that the probability of the ball ending up inside the goal will be high enough, independent of the particular goalkeeper behavior.

We have incorporated our scoring policy into the agent's main decision loop as follows. When the agent has control of the ball, he first checks whether the probability of scoring is larger than a certain threshold. This threshold is represented by the parameter *ScoringProbThr* which has a value of 0.9 in our current implementation. If the total scoring probability exceeds the threshold then the agent tries to score by shooting the ball with maximum power towards the best scoring point. Otherwise he considers different options, such as passing to a teammate or dribbling with the ball, which he performs when the predicted success rate is high enough. However, when all possible alternatives fail and the agent stands at a close distance to the goal, he decides to shoot to the best scoring point anyhow.

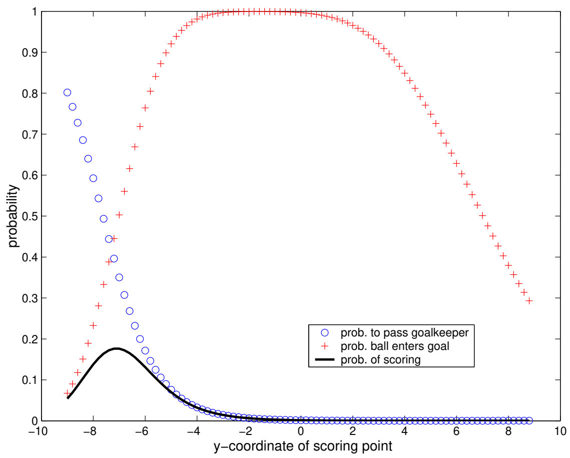
Figure 8.6 shows two successive situations which were taken from a real match played by *UvA Trilearn* (yellow team). In Figure 8.6(a) the player with the ball stands to the left of the goal which is covered



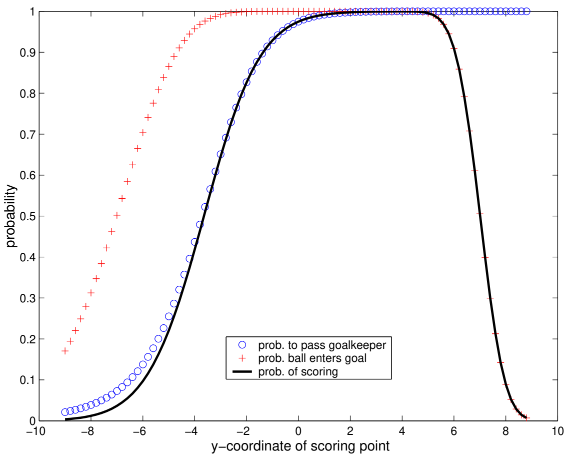
(a) Goalkeeper covers goal well. Through pass is given.



(b) Goalkeeper outplayed. Good chance to score.



(c) Low scoring probability for all goal points.



(d) High scoring probability in right half of goal.

Figure 8.6: Two successive match situations together with the associated scoring probability curves.

well by the opponent goalkeeper. This player therefore decides to give a through pass to the right wing attacker who is in a better position. Figure 8.6(b) shows the situation a few cycles later. The right wing attacker has now intercepted the ball and has a good chance to score. The scoring probability curves which correspond to these situations are shown in Figures 8.6(c) and 8.6(d). In these figures the horizontal axis represents the y-coordinate on the goal line to which the ball is shot. Note that the left and right goalposts are respectively located at y-coordinates -7.01 and 7.01. Figure 8.6(c) shows that in the first situation the total scoring probability (solid line) is very low for all the points on the goal line[8] and the player with the ball thus rightly decides not to shoot to the goal. However, several cycles later the situation is completely different. The right wing attacker now has a high probability of scoring in the right half of the goal due to the fact that the goalkeeper stands too far to the left. In Figure 8.6(d) the right slope of the total scoring probability is bounded by the probability that the ball enters the goal, whereas the left slope is bounded by the probability that the goalkeeper intercepts the ball. Since the total scoring probability equals 1.0 for y-coordinates around 3.0, the striker decides to shoot there (and scores as a result).

The scoring policy described in this chapter was used by *UvA Trilearn* at the *RoboCup-2001* robotic soccer world championship. Table 8.1 shows statistics concerning the percentage of successful scoring attempts for the top four teams in this tournament. The percentages are based on all the matches that were played by these teams during the second group stage and double elimination stage of the competition. The results show that during these stages the success rate for *UvA Trilearn* was higher than for the other three teams. It must be noted however, that it is difficult to compare the percentages due to the fact that the different teams use different attacking strategies. Furthermore, the statistics are based on different matches against different opponents and the decision whether to shoot to the goal or not is likely to be based on different factors for each team. The *Brainstormers*, for example, might try to score even when the probability of success is moderate. It is impossible however, to deduce the intentions of other players by observing their behavior and these factors have therefore not been incorporated into the results. Nevertheless, the results show that the *UvA Trilearn* agents manage to score a high percentage of their attempted goal shots.

Team	**Attempts**	**Success**	**Percentage**
Tsinghuaeolus	70	56	80.00%
Brainstormers	39	23	58.97%
FC Portugal	93	72	77.42%
UvA Trilearn	42	34	80.95%

Table 8.1: Percentage of successful scoring attempts for the top four teams at *RoboCup-2001*. These statistics were generated by *RoboBase*, a logplayer and analysis tool for RoboCup logfiles [87].

8.6 Conclusion

In this chapter we have described a methodology that allows a simulated soccer agent to determine the probability of scoring when he shoots the ball to a specific point in the goal in a given situation. The single probability that the ball enters the goal (first subproblem) depends on the values of various server parameters which control the movement noise of the ball, the size of the goal, etc. The approach presented in Section 8.2 is general in the sense that it enables one to 'learn' this probability even when these server parameter values change (e.g. in a future version of the server). However, the single probability of passing the goalkeeper (second subproblem) depends on the opponent goalkeeper and different goalkeepers

[8] Note that the noise in the ball movement causes a non-zero scoring probability when the ball is shot to a point just outside the goal. In our implementation these points are never selected however, since we only consider points on the goal line for which the single probability of entering the goal is more than 0.7 (=*EnterGoalThr*).

exhibit different behaviors. In our current implementation, we have based this probability entirely on the goalkeeper of *RoboCup-2000* winner *FC Portugal*. Since this is a good goalkeeper, the approach presented in Section 8.3 is useful against other goalkeepers as well. Ideally however, the probability of passing the goalkeeper should be *adaptive* and the model should incorporate information about the current opponent goalkeeper instead of using that of a particular team. The desired case would be to let the model adapt itself *during* the game, using little prior information about the current goalkeeper. This is a difficult problem because learning must be based on only a few scoring attempts. It is therefore important to extract the most relevant features and to parametrize the intercepting behavior of the opponent goalkeeper in a compact manner that permits on-line learning (e.g. through the use of statistics collected by the coach).

Another drawback in our solution to the second subproblem is that the set-up for the learning experiment is such that it includes a fixed distance to the scoring point. Consequently, Equation 8.10 actually reflects the probability that the ball will travel a distance of a least 20 meters before it is intercepted by the opponent goalkeeper. In the current setup this result cannot be generalized for different distances. When the distance to the scoring point is smaller, for example, the goalkeeper might be able to intercept the ball before it has traveled 20 meters but not before it crosses the goal line. The returned probability of passing the goalkeeper will thus always be too low in these cases. Conversely, the returned probability will always be too high when the distance to the scoring point is larger: it is possible that the goalkeeper intercepts the ball before the goal line after it has traveled more than 20 meters.

A better solution would clearly be to adapt the learning experiment by randomly placing the ball at different distances to the goal and to incorporate an extra feature into the model which represents the distance to the scoring point. This will slightly complicate the problem however, since the separation between the two classes will no longer be linear. We have therefore chosen to base the classification on two features only (angle and distance to the goalkeeper) and to fix the distance to the scoring point at 20 meters in our experiment. Note that the resulting model yields a good approximation since this distance is about equal to the average shooting distance. More importantly, this distance also provides a clear cut-off point between two distinct classes of situations. The first class represents samples in which the goalkeeper is able to intercept the ball quickly before it has traveled a large distance. This happens, for example, when the goalkeeper stands close to the ball trajectory. When this is not the case however, the high speed of the ball resulting from the kick usually causes it to move past the goalkeeper. As a result, the goalkeeper must run after the ball and can only catch up with it when its speed has decreased sufficiently. In these cases, which are represented by the second class, the goalkeeper will need more time to intercept the ball and during this time the ball travels a large distance. It is important to realize that nearly every scoring attempt which occurs during a match belongs to one of these classes. Moreover, we empirically found that a traveled distance of about 20 meters clearly separated one class from the other. We can therefore expect that (8.10) provides a good approximation of the non-interception posterior probability.

Chapter 9

Team Strategy

In Chapters 6 and 7 we have discussed the most important aspects of an individual agent: the world model he creates based on his perceptions and the skills he can perform. The behavior of an agent can be defined as the bridge between these two components. It determines which skill is selected in a given situation based on information from the world model. In a robotic soccer game however, each agent is part of a team which consists of 11 agents that must cooperate to achieve a common goal. Although perception and action are local for each agent, they should thus also be part of a larger collaborative plan which is shared by all the teammates. In this chapter we describe the team strategy of the *UvA Trilearn 2001* soccer simulation team. We will focus on team behaviors and address various issues concerning the coordination and cooperation between individual agents. The chapter is organized as follows. In Section 9.1 we provide a general introduction and list the most important aspects of a multi-agent soccer strategy. Team formations are introduced in Section 9.2 together with the strategic positioning mechanism which is used by the agents. In Section 9.3 we discuss our use of heterogeneous players followed by a description of the *UvA Trilearn* communication model in Section 9.4. Section 9.5 is devoted to the topic of action selection. Since the development of the action selection procedure for the agents has been an incremental process, this procedure is explained for the *UvA Trilearn* team by means of several intermediate teams from which this team has evolved. Results showing the effectiveness of the implementations described in earlier sections are presented in Section 9.6. Section 9.7 contains a number of concluding remarks.

9.1 Introduction

The *strategy* of a team of agents can be defined as the collective plan which is followed in trying to achieve a common goal with the available resources. For a soccer team this amounts to the way in which the different players in the team should act to accomplish their common goal of winning the game. Once a soccer agent is able to execute certain individual skills he must learn to act strategically as part of the team. This means estimating the long-term effects of actions in the context of a soccer game and selecting a skill which best serves the purpose of the team. The behavior of each individual agent constitutes a mapping from perceptions to actions: each agent builds a world model based on his perceptions and uses this model to select an appropriate action in a given situation. The decision which skill to execute depends on the strategy of the team which can be seen to define the way in which the individual agent behaviors are coordinated. As a result, cooperation between the agents will emerge from this strategy. We consider the most important aspects of a multi-agent soccer strategy to be the following:

- The strategy must specify the formation of the team and the position of the agents inside this formation. Furthermore, it can define which formations are appropriate for which game situations.
- The strategy must define different roles inside a formation and assign these roles to various player positions. It should also indicate which kinds of heterogeneous players are useful for which roles.
- For each type of player (defender, midfielder, etc.) the strategy must specify the behavior which is associated with his current role. A defensive player, for example, should play more conservatively than an attacking player and as a result should consider different actions.
- The strategy must incorporate information about how an agent should adapt his behavior to the current situation. The action that an agent chooses, for example, should depend on which part of the field he is currently located and on the positions of other teammates and opponents.
- The strategy must specify how to coordinate individual agent behaviors. This is important since team members may otherwise not agree on the current strategy or on the mapping from teammates to roles within this strategy. Although the results will not immediately be disastrous if different agents temporarily adopt different strategies, the team members are more likely to achieve their goal if they can stay coordinated. Possible ways to achieve this coordination are via pre-defined information which is known to all the teammates and through the use of communication.
- The strategy must indicate how each player should manage his stamina during a game. For example, when a player is tired he should not make a run for the ball unless this is absolutely necessary.

Furthermore, the strategy of a team of agents can also be influenced by various external factors. Some of these factors which should be considered are the following:

- The strength of the opponent team. A strategy which is successful against a weak opponent will not necessarily work against a stronger team. An important difference in this respect is the accuracy with which certain actions must be performed. When the opponent players are good at intercepting the ball, for example, it is important that the ball is passed accurately if the team wants to keep possession. Against a weaker team however, it is often possible to pass with more risk.
- The type of opponent (i.e. their playing style). Against an offensive team one should adopt a different strategy than against a defensive team.
- The state of the game. The strategy of the team can be related to the current score. For example, if the team is leading towards the end of the game it is possible to switch to a more defensive strategy in order to preserve the advantage. If the team is trailing near the end however, it could decide to start taking more risks in an attempt to make up the deficit.
- The current game situation. A team should focus on their defense if the opponent team is building up a dangerous attack, whereas the team's behavior can be more aggressive when a teammate has possession of the ball inside the opponent's penalty area.
- The available resources. Since the *soccer server* makes use of randomly generated heterogeneous players, it is important to adapt the team strategy to the current player types. Depending on the available player types it must be decided which types are selected and which role they must fulfill inside the current formation. Furthermore, if it turns out that certain players are performing badly (e.g. become tired too quickly) they must be substituted. Note that these choices can be based on the current state of the game or on the playing style of the opponent team.

The problem that will be addressed in this chapter is how to incorporate these aspects into a common strategic framework. It must be noted however, that our initial decision to build the team from scratch has forced us to spend the majority of our time on the implementation of low-level and intermediate level issues such as perception and actuator control, world modeling and skills execution. As a result, the high-level team strategy has received less attention and although it contains all the necessary components to put up a working soccer team, the setup is not as profound as that for the lower levels. The main principle is that we make a distinction between *active* and *passive* situations for each agent depending on whether he currently has an active role in the game or not. In general, a player is in an active situation when he is located near the ball and in a passive situation otherwise. If an agent is in a passive situation he moves to a strategic position on the field in anticipation of the possibility of becoming active again. This position is based on the current team formation, the role of the agent inside this formation and the position of the ball on the field (see Section 9.2). For certain roles inside the formation we use heterogeneous player types for which the characteristics are suitable to these roles (see Section 9.3). If an agent is in an active situation he chooses an action based on his current position and on the position of other players on the field. For reasons of simplicity the agents do not explicitly take the strength of the opponent team into account when selecting an appropriate action.

Active as well as passive agents determine a new action in each cycle and do not commit to a previous plan. This is because a soccer game provides a highly dynamic environment in which a premeditated action sequence will have to be adapted significantly as soon as the opponents counteract it. We have therefore chosen to base the action selection procedure (see Section 9.5) only on the current state of the environment. Furthermore, the individual agent behaviors are coordinated implicitly via pre-defined information which is known to all the teammates. In [90] this kind of information is referred to as the *locker-room agreement*. There is no explicit coordination between the agents in the form of inter-agent communication because the *soccer server* communication channel has a low bandwidth and is extremely unreliable (see Section 3.2.2). It is therefore important that the agents do not depend on communication when an action has to be performed. In our current implementation, the agents only use communication to increase the amount of up-to-date information in their world model (see Section 9.4) and this significantly improves their ability to cooperate with other teammates.

9.2 Formations and Strategic Positioning

Collaboration between agents can be achieved by making use of *formations*. A formation can be seen as a specified arrangement of a group of agents which decomposes the task space by defining a set of roles. Each role inside a formation can be filled by a single agent although the roles themselves are specified independently from the agents that are to fill them. Formations are a principal concept for a soccer team since they take care of a good distribution of players on the field. When a formation is used, the players will be able to keep the most important parts of the field well covered during a match thereby avoiding a clustering of team members in a certain area (e.g. around the ball as is usually the case in 'kiddie soccer' where no formations are used). In our implementation, formations deal with the positioning of agents during passive situations, i.e. during situations in which an agent does not have an active role in the game. Here we regard every situation in which an agent is not located close to the ball as passive. We have largely based our positioning mechanism on a method called *Situation Based Strategic Positioning* which has been introduced by *FC Portugal* [56, 77]. This means that formations define a set of roles which consist of a player type (e.g. wing attacker) and a home position on the field. Inside the current formation each agent is assigned a role which he must fulfill for as long as this formation is used. In passive situations an agent determines a strategic position on the field by computing a weighted sum of his home position and the current position of the ball which serves as an attraction point. It is important

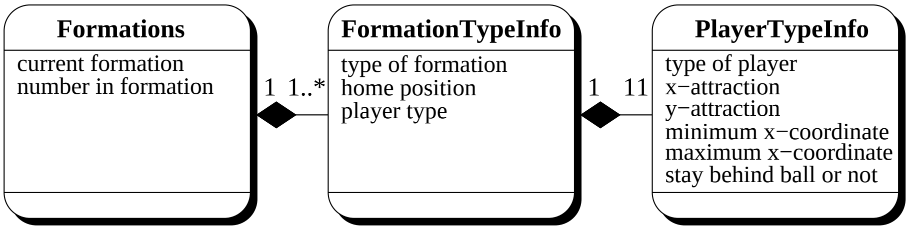


Figure 9.1: UML class diagram of the classes related to formations and positioning

to note that the attraction to the ball is different for each player type. The various home positions can thus be seen to define the positioning pattern between the teammates, whereas the ball attraction defines the way in which this pattern is stretched over the field in a given situation.

The implementation of our positioning mechanism consists of three separate classes which together contain all the information for various types of formations. Figure 9.1 shows an UML class diagram of these classes and their relations. For reasons of space and clarity the class methods have been omitted. The contents of the three classes are explained in more detail below.

- *Formations*. This class contains a collection of *FormationTypeInfo* objects which hold all the necessary information for the different types of formations. Furthermore, it stores the current team formation and the agent's number in this formation which determines his role. The method which is used to determine a strategic position on the field is also contained in this class.
- *FormationTypeInfo*. This class contains all the information about one specific formation. It stores the type of this formation as well as the agent's home position in the formation and his player type (e.g. wing defender, central midfielder, etc.). Furthermore, it contains a collection of *PlayerTypeInfo* objects which hold all the necessary information for the different player types.
- *PlayerTypeInfo*. This class contains all the information about one specific player type. It stores this type as well as a number of values which are used to determine a strategic position for such a player. These are the following:
  - The attraction to the ball in x-direction and y-direction. These are values from the interval $[0, 1]$ which determine the weight of the ball position when determining a strategic position.
  - The minimum and maximum allowed x-coordinates of a strategic position for this player type.
  - A boolean indicating whether a player of this type is allowed to move to a strategic position in front of the ball. This is not allowed, for example, for a central defender (also called 'sweeper').

When an agent is initialized, the information for different types of formations is read from a configuration file and stored in the appropriate class. Each agent reads the same file and therefore knows the roles and player type information for all his teammates. As a result, each player can uniquely determine the strategic position for his teammates given the current ball position. This type of pre-defined information that is used by all the team members is an example of a *locker-room agreement*. It is used, for example, to determine which player has to perform a free kick when one has been awarded. In our implementation this is not the player which stands closest to the ball at present, but the player for whom the strategic position is closest to the ball. In this way, the formation will stay more intact during a dead ball situation. Since each player is able to determine the strategic position for all his teammates, he can thus determine which teammate (including himself) should take the free kick without having to resort to unreliable communication. The method which is used to compute the strategic position (s_x, s_y) for a player is shown in Algorithm 9.1.

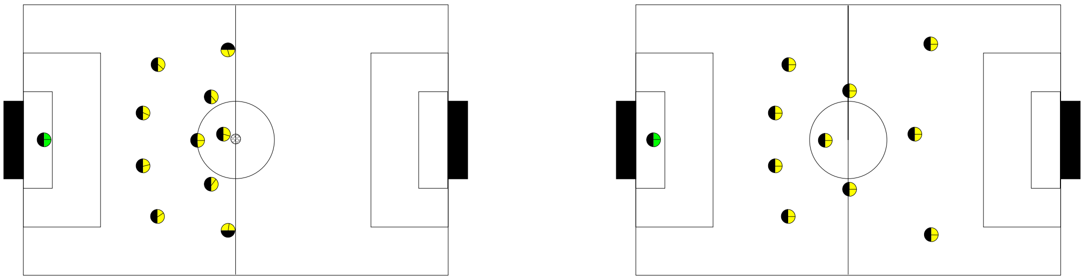
```
getStrategicPosition(q⃗)      // q⃗ = current ball position

get home position (p_x, p_y) and ball attraction factors (attr_x, attr_y)
get x-coordinate range [min_x, max_x] and boolean BehindBall
(s_x, s_y) = (p_x, p_y) + (attr_x, attr_y) · (q_x, q_y)
if BehindBall == true and s_x > q_x then
   s_x = q_x
end if
if s_x > max_x then
   s_x = max_x
else if s_x < min_x then
   s_x = min_x
end if
return (s_x, s_y)
```

Algorithm 9.1: Method for determining the strategic position (s_x, s_y) of a player.

It is important to realize that during a match various situations can occur in which the agent should not move towards the strategic position as returned by the `getStrategicPosition` method. An example of such a situation is when the returned position is *offside*. A player is in an offside position if he stands in front of the ball on the opponent's half of the field and closer to the opponent's end line than all or all but one[1] of the opponent players when the ball is passed to him. To avoid being caught offside, a player must always stay behind a so-called 'offside line' which is determined by the opponent player with the second-highest x-coordinate or by the ball if the ball position is further forward than this player. If it turns out that the returned strategic position lies past the offside line then the x-coordinate of this position is adjusted accordingly (i.e. to an onside position). Another example of a situation in which the strategic position returned in Algorithm 9.1 might be illegal is when the opponent team has been awarded a goal kick[2]. In this case, it is not allowed to enter the opponent's penalty area until the ball has left this area. If an agent's strategic position (s_x, s_y) is located inside this area the value for s_x is therefore adjusted.

In our current implementation we only use two types of formations: an initial formation which is used before the start of a half and after a goal has been scored and a 4-3-3 formation[3] which is used during



(a) Home positions in initial formation

(b) Home positions in 4-3-3 formation

Figure 9.2: Home positions on the field in the two formations used by *UvA Trilearn*.

[1] This 'one' is usually the opponent goalkeeper.

[2] A goal kick is awarded to the defending team if the ball goes out of bounds over the end line and was last touched by the attacking team. If the ball was last touched by the defending team then the attacking team is awarded a corner kick.

[3] Soccer formations are typically described as A-B-C where A, B and C are the numbers of defenders, midfielders and forwards respectively. It is assumed that the 11th player is the goalkeeper. [54]

	Number in Formation										
	1	2	3	4	5	6	7	8	9	10	11
home_x	−50.0	−13.0	−14.0	−14.0	−13.0	−5.0	0.0	0.0	15.0	18.0	18.0
home_y	0.0	16.0	5.0	−5.0	−16.0	0.5	11.0	−11.0	−0.5	20.0	−20.0
pl_type	1	3	2	2	3	4	5	5	7	6	6
attr_x	0.1	0.6	0.6	0.6	0.6	0.6	0.6	0.6	0.6	0.5	0.5
attr_y	0.1	0.35	0.25	0.25	0.35	0.25	0.3	0.3	0.2	0.25	0.25
beh_ball	1	0	1	1	0	0	0	0	0	0	0
min_x	−50.5	−45.0	−45.0	−45.0	−45.0	−30.0	−30.0	−30.0	−2.0	−2.0	−2.0
max_x	−30.0	35.0	0.0	0.0	35.0	42.0	42.0	42.0	40.0	42.0	42.0

Table 9.1: Complete specification of the 4-3-3 formation used by *UvA Trilearn*. The player type numbers denote the following types: 1=goalkeeper, 2=central defender (sweeper), 3=wing defender, 4=central midfielder, 5=wing midfielder, 6=wing attacker and 7=central attacker.

normal play. Figure 9.2 shows the home positions of the agents in these two formations. A complete specification of the 4-3-3 formation, including ball attraction factors and x-coordinate ranges for each player type, is given in Table 9.1. We have chosen to use this 4-3-3 formation as our standard formation based on an analysis of logfiles from previous matches. This analysis showed that it was very difficult to pass a good goalkeeper when the attack originated from the center of the field. The reason for this was that the goalkeeper then stood in front of the center of his goal and was able to save most shots on goal since he only had to travel a small distance before he could catch the ball. However, when the attack originated from the wings the goalkeeper had to move to the side of the goal to cover the scoring angle. A quick pass from the wing attacker to the central attacker then usually outplayed the goalkeeper (who would have to travel a large distance to cover his goal again) and led to a good scoring opportunity. Attacking from the wings thus proved to be effective and we therefore decided to use a 4-3-3 formation which is the standard formation for using wing attackers [54]. Besides the initial formation shown in Figure 9.2(a), we have chosen to specify only this 4-3-3 formation due to time constraints. In this way, we were able to concentrate on this specific formation and make sure that the team as a whole played well in it. This has led to good results in both defensive and offensive play as will be shown in Section 9.6.

9.3 Heterogeneous Player Selection

A group of agents is called *homogeneous* if the agents are physically and behaviorally identical and *heterogeneous* if they are different in some way. The introduction of heterogeneous players was a new feature in *soccer server* version 7. In earlier versions, all the players on the field were homogeneous and the player parameters had the same values for each player. As of version 7 however, each team can choose from several different player types which have different characteristics. These player types are randomly generated when the server is started. The default player type is always the same, whereas the other types are different on each server restart. The characteristics of these non-default players are based on certain trade-offs with respect to the player parameter values used by the default player type. In the current server implementation five trade-offs have been defined for specific combinations of player parameters. These trade-offs are shown in Table 9.2. In each case one of the parameters in the combination gets an improved value as compared to the default, whereas the other becomes worse. The actual values for these player parameters are randomly chosen from different intervals which are defined in the server. For an exact description of these value ranges and the mentioned trade-offs we refer the reader to Table 3.13.

improvement	weakness	trade-off description
player_speed_max	stamina_inc_max	player is faster, but his stamina recovery is slower
player_decay	intertia_moment	player's speed decays slower, but he can turn less
dash_power_rate	player_size	player can accelerate faster, but he is smaller
kickable_margin	kick_rand	kickable distance is larger, but noise is added to kick
extra_stamina	effort_min,effort_max	player has more stamina, but dash is less efficient

Table 9.2: Trade-offs between player parameters for heterogeneous players.

The coach of the team is responsible for selecting which player types to use and for substituting players when necessary. When the server is started all 11 players are initialized to default players. The coach is then allowed to change each player to a non-default player with the restriction that he is not permitted to put more than three non-default players from the same type on the field simultaneously. Which player types are selected should depend on the current strategy of the team. Within this strategy certain player types will be more useful for a particular role than others. We have already seen in Section 9.2 that the *UvA Trilearn* team always uses a 4-3-3 formation which is the standard for attacking along the wings. In such a formation, the wing attackers must be able to move fast in order to cut through the enemy defense and outrun the opponent defenders before passing the ball to a central attacker in front of the goal. Furthermore, these players must be capable of sustaining a long sprint towards the opponent's end line without getting too tired. The policy which is used for heterogeneous player selection has been based on these characteristics of the 4-3-3 formation. In order to determine which player types are best suited for certain roles, we have developed a utility function which returns a numeric value for each type based on the player parameters. This value represents a measure for the quality of a particular player type with respect to a number of properties which are crucial for successfully filling these roles inside a 4-3-3 formation. For each heterogeneous player type i, the utility function considers the following characteristics:

- The maximum speed m_i for a player of type i. This value is represented by the player parameter `player_speed_max` and denotes the maximum distance that a player can cover in one cycle (neglecting noise and wind). If a player is capable of reaching a higher maximum speed he will thus be able to move faster which is an important characteristic for a wing player in a 4-3-3 formation.

- The amount of stamina l_i that a player of type i loses in one cycle when he moves at maximum speed. This value gives an indication of how long a player will be able to maintain his maximum speed while running. Recall from Section 3.4.2 that the amount of stamina which is lost when a player dashes depends on the power which has been supplied to the **dash** command: for a forward dash the stamina loss equals this power, whereas it equals twice the absolute value of this power when the player dashes backwards. Furthermore, a player's stamina gets restored by `stamina_inc_max` in each cycle until it has reached `stamina_max`. The dash power d_i which is required for a player of type i to maintain his maximum speed can be calculated according to the following formula[4]:

$$d_i = \frac{\texttt{player\_speed\_max} \cdot (1.0 - \texttt{player\_decay})}{\texttt{effort\_max} \cdot \texttt{dash\_power\_rate} \cdot \texttt{max\_power}} \cdot \texttt{max\_power} \qquad (9.1)$$

  Here the numerator of the fraction denotes the amount of speed which is lost during a cycle as a result of speed decay (assuming that the player is already moving at maximum speed) and the denominator denotes the maximum amount of speed which can be added in one cycle when dashing with maximum power. If we multiply the ratio between these values by the maximum power which can be supplied to a **dash** command we get the dash power d_i which is required to compensate for

[4] Note that we have deliberately not eliminated the two occurrences of `max_power` in this formula to keep the result intuitive.

the lost speed. Note that if the numerator in the fraction is larger than the denominator (i.e. more speed is lost during a cycle than can be gained by dashing) then it is not possible for the player to reach the maximum speed since the dash power which is required to achieve this exceeds the maximum. The value for d_i can be used to calculate the amount of stamina l_i that is effectively lost during a cycle if maximum speed is to be maintained. This amount l_i equals the stamina which is lost as a result of the **dash** minus the amount which is added in each cycle, i.e.

$$l_i = d_i - \texttt{stamina\_inc\_max} \tag{9.2}$$

For each heterogeneous player type, the utility function combines the maximum speed m_i with the stamina loss l_i to obtain a utility value for this type. This is done by creating a simple ordering for both characteristics: the maximum speeds for each player type are listed in ascending order, whereas the stamina losses are listed in descending order. Note that the lists are thus ordered from worst to best, i.e. the higher the index in the list the better. To each player type i we then assign two numbers, μ_i and λ_i, which denote their index in the sorted lists for maximum speeds and stamina losses respectively. This gives:

$$\mu_i = 1 + \sum_{j=1}^{n} \Theta(m_i > m_j) \quad \text{and} \quad \lambda_i = 1 + \sum_{j=1}^{n} \Theta(l_i < l_j) \tag{9.3}$$

where $\Theta(\beta)$ is a function that returns 1 when the boolean expression β is true and 0 otherwise. The sum $\mu_i + \lambda_i$ represents a measure for how well a player is capable of moving at a high speed while not consuming too much stamina in the process. The player type for which $\mu_i + \lambda_i$ has the highest value is therefore assumed to be the most suitable for playing on the wings in a 4-3-3 formation. Note that it is possible that the utility function returns the highest value for the default player type. In this case, all the players in the team will remain default players. Otherwise, the default wing attackers are replaced by the non-default player type with the highest utility value. The same is done for the central attacker since this player must also be capable of running fast in order to keep up with the wing players during an attack and to have a chance of receiving their cross pass. Furthermore, the wing defenders are changed to the player type with the second-highest utility value if this value is higher than that for the default type. This is done to create a better defense against a possible wing attack by the opponent team. Note that we cannot use the best type for these players since we have already used this type for three players on the field which is the maximum for a non-default type. The other players in the formation are kept unchanged (i.e. default) due to the fact that the higher stamina loss for non-default types is a big disadvantage for a midfielder or central defender. In our current implementation, the player types which are assigned to each role stay the same throughout the match. The coach thus makes no substitutions during the game. The reason for this is that the decision when to substitute a player requires some form of multi-agent modeling by the coach which has currently not been implemented.

9.4 Communication Model

Agents in the *soccer server* are not allowed to communicate with each other directly, but only indirectly through the use of server-defined **say** and **hear** protocols which restrict the communication. The *soccer server* communication paradigm models a crowded, low-bandwidth environment in which the agents from both teams use a single, unreliable communication channel [90]. Spoken messages have a limited length (512 characters) and are only transmitted to players within a certain distance (50 meters) from the speaker. No information is given about which player has sent the message or about the distance to the sender. Furthermore, the aural sensor has a limited capacity. Each player can hear only one message from a teammate every two simulation cycles and any extra messages that arrive during this time are discarded.

One can thus never be sure that a spoken message will actually reach the intended receiver. Because of this unreliability it is important that the agents do not depend on communication when they have to perform an action. In our current implementation, the agents therefore only use communication to help teammates improve their knowledge about the state of the world. Depending on the position of the ball, the agent that has the best view of the field communicates his world model to all nearby teammates. The teammates that hear this message can then use the communicated information about the parts of the field which are currently not visible to them to elucidate some of their hidden state and increase the reliability of their world model. As a result, the amount of up-to-date information in the world model of the agents will increase significantly and this extra information can also be taken into account during the action selection process. The extra information is not necessary however, since without it the agents will also be able to maintain a reasonable approximation of the world state and determine an appropriate action. This is important since it makes the agents robust to lost messages.

The *UvA Trilearn* communication model defines which player should speak in a given situation and how frequently the broadcasting should take place. During the game, each player in the team determines whether he is currently the best player to communicate his world model. This decision depends on the position of the ball on the field as well as the role of the player inside the current team formation. Since the main objective of our communication model is to increase the amount of up-to-date information in the world model of the agents, it is important that the communicating player has a good view of the field and sees a lot of teammates and opponents. Especially the part of the field where the ball (and thus the action) is located should be clearly visible to this player. We empirically found that on average the midfielders at the wings have the best view of the field since they are usually located around the center line and close to the side where they can see most of the action when facing the ball. In our current implementation, the wing midfielders are therefore the ones to communicate their world model. Note that, besides their good view of the field, the central position of these players brings two additional advantages. Firstly, most teammates will be located within the hearable distance of 50 meters from the speaker which means that the team can take full advantage of the communication. Secondly, most objects in the speaker's view cone will on average be quite close to him which increases the precision of his visual information (see Section 3.2.1) and thus the reliability of the world model he communicates.

Algorithm 9.2 shows the method which is used by the agents to determine whether they should communicate their world model. If the ball is located on the right side of the field (positive y-coordinate) the left midfielder is selected since he has a good view of a large playing area when he faces the ball. Furthermore, this player currently does not have an active part in the game and this allows him to trade off the frequency of visual information against the width of his view cone. Figure 9.3 shows the visible area of a left midfielder when facing the ball on the right half of the field (bottom half of the picture) with a view cone of 180 degrees. The wider view cone enables this player to see a larger part of the field and as

<u>`shallISaySomething()`</u>

if I haven't spoken for two cycles **then**
  // $\vec{q}$ = current ball position
  if $q_y > 0$ **and** my role == left midfielder **then**
    communicate my worldmodel to teammates
  else if $q_y \leq 0$ **and** my role == right midfielder **then**
    communicate my worldmodel to teammates
  end if
end if

Algorithm 9.2: Method for determining whether a player should communicate his world model.

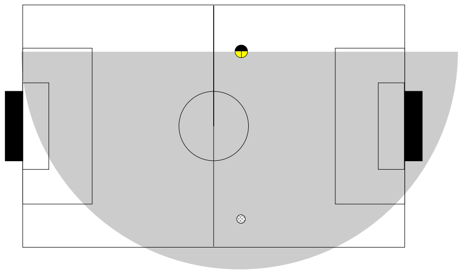


Figure 9.3: Visible area (shaded) of a left midfielder when he faces the ball on the right side of the field. The player uses a wide view cone which enables him to see most of the playing area.

a result he can communicate more information to his teammates. He will receive visual information less frequently however, but since his role in the game is currently a passive one this is acceptable. In the same way, the midfielder on the right communicates his world model if the ball is located on the left side of the field (negative y-coordinate). Note that the method presented in Algorithm 9.2 guarantees that only one agent will communicate his world model every two cycles. As a result, we can be sure that every message is received by all the players since there are no other messages to overload the communication channel. The possibility that one of the wing midfielders might not be able to communicate for some reason is not taken into account since this never occurs during regular matches.

9.5 Action Selection

In order to satisfy his goals, an autonomous agent must select at each moment in time the most appropriate action among all possible actions that he can execute. This is what is known as the *Action Selection Problem* (ASP). In the context of ASP, an *Action Selection Mechanism* (ASM) can be regarded as a computational mechanism that must produce a selected action as output given various external and/or internal stimuli as inputs. While the ASP refers to *which* action an agent must select in a given situation, the ASM thus specifies *how* these actions are selected.

Since a soccer game provides a highly dynamic environment it is difficult to create a fixed attacking plan or any other form of premeditated action sequence that works in every situation. As soon as the opponents move to counteract the attack, for example, the plan must be adapted to the changed environment. Instead of a sequence of actions, the *UvA Trilearn* agents therefore determine just a single action in each cycle and base their action choice only on the current world state. In this section we will present our solution to the ASP and describe the ASM which is used by our agents. Since it is impossible to create a complete action selection strategy in one step, the development of the action selection procedure for the agents has been an incremental process. The first iteration consisted of a simple decision procedure which was used to test the low-level performance and basic agent skills. After testing the performance of the resulting team, we gradually extended this version by introducing more advanced skills and creating a more sophisticated strategy. The main advantages of this approach were that we had a working system at all times and that we always knew that occurring problems had to originate from the last refinement step. Furthermore, the effects of added functionalities could be tested by playing against previous versions of the team which did not contain them. We will describe the action selection procedure for the *UvA Trilearn* team with the help of several intermediate teams from which this team has evolved. Apart from the first team, each team that is described in this section was either used to qualify for or participate in an international competition.

9.5.1 First Version: De Meer 5

During the initial stages of the project we spent the majority of our time on solving low-level and intermediate-level problems such as agent-environment synchronization and world modeling. In order to create a 'real' team however, we needed some kind of high-level decision procedure. The first complete version of our team that could actually play a game of soccer was ironically called *De Meer 5*.[5] The idea for this team was motivated by a simple team which had been released by the creators of *RoboCup-2000* winner *FC Portugal 2000*. This basic team, which we will refer to as *Simple Portugal*, consisted of a simple high-level decision procedure on top of the low-level implementation of 1999 world champion *CMUnited-99*. It was claimed that this team would be able to beat many of the teams that had participated at *RoboCup-2000* and as such would provide a good starting point for new teams to practice against. *De Meer 5* consisted of a basic if-then-else action selection mechanism which was almost equal to the high-level procedure used by *Simple Portugal* and which had been put on top of our own low-level implementation. It was our intention to test the implementation of *De Meer 5* by playing against *Simple Portugal*. Since the high-level decision procedure for both teams was the same, it was clear that the lower levels would make the difference. Playing against *Simple Portugal* would thus be a good test of the quality of our low-level implementation, since it effectively meant comparing our lower levels to those of *CMUnited-99*.

Algorithm 9.3 shows the action selection procedure which is used by the agents of the *De Meer 5* soccer simulation team. If the confidence associated with the current ball position drops below a certain threshold, a *De Meer 5* agent searches for the ball. Otherwise, he determines whether he currently has an active part in the game by checking if the ball is located within his kickable range or if he is the fastest teammate to the ball. In the former case he kicks the ball with maximum power to a random corner in the opponent's goal regardless of his position on the field and in the latter case he will start to intercept. If the agent's role in the game is currently a passive one he moves to a strategic position which is based on the 4-3-3 formation described in Section 9.2 and when he is already close to this position he simply turns towards the ball. Furthermore, we implemented a goalkeeper that uses a slightly different decision loop than the regular field players. The strategic position for this goalkeeper is determined by defining a rectangle in front of the goal and taking the first intersection point (i.e. closest to the ball) of this rectangle and the line that runs from the ball to the center of the goal line. If the goalkeeper is the fastest player to the ball and if the point of interception is located inside his own penalty area he tries to intercept the ball. Note that the goalkeeper that we implemented exhibited similar behavior to the one used by *Simple Portugal*. In this way, the difference between both teams would still be the low-level implementation only.

```
if confidence associated with current ball position is too low then
  search ball
else if ball is kickable then
  kick ball with maximum power to random corner in opponent goal
else if fastest teammate to ball is me then
  intercept ball
else if distance to strategic position > 1 then
  go to strategic position
else
  turn towards ball
end if
```

Algorithm 9.3: Action selection procedure for soccer simulation team *De Meer 5*.

[5] The father of one of the authors actually plays in a team which has the same name. This name was chosen because the team's weekly performance failed to impress the corresponding author as did the performance of our first working version.

Although the action selection procedure which is used by the *De Meer 5* agents is very simple, it contains the three most important skills for any soccer player: kicking, intercepting and positioning. We have tested the implementation of *De Meer 5* by playing against the *Simple Portugal* team. Initially, *De Meer 5* performed very badly against this team. This was mainly because our intercept method was inferior to that of our opponents. The *Simple Portugal* players always succeeded in intercepting the ball quickly, whereas our players often failed to intercept correctly or did so at a suboptimal interception point. We have therefore spent a long time on improving this particular skill (see Sections 7.3.4 and 7.4.1) and after a while we were able to beat the *Simple Portugal* team by a combined score of 20-17 over 10 full-length matches. At that point, it could thus be concluded that our overall low-level implementation was about as good as that of *CMUnited-99* and we decided to concentrate on the next version of our team.

9.5.2 UvA Trilearn Qualification Team for RoboCup-2001

To qualify for *RoboCup-2001* we had to provide a logfile of a full-length game between *UvA Trilearn* and *FC Portugal 2000*, the winning team at *RoboCup-2000*, and a 2-page research abstract describing the research focus and scientific contributions of our team [20]. At this stage, *De Meer 5* was clearly not good enough to beat the previous champion and since we had little time to produce a winning logfile we estimated that our best chance of qualifying would be to temporarily tune our high-level strategy completely to playing well against *FC Portugal*. A quick visual analysis of this team revealed that their wing attack was very strong and that they passed the ball very accurately. However, it seemed that their attack through the center was not so effective and that defending was not their strongest point. In order to play well against this team we thus had to be able to defend their wing attack and test their defense.

The action selection procedure for our qualification team is almost identical to that for *De Meer 5* apart from when the agent has control of the ball. In this situation, a *De Meer 5* agent always kicks the ball with maximum power to a random corner in the opponent's goal regardless of his position on the field. It was clear however, that in order to qualify our agents needed a more sophisticated strategy in which they considered alternative options when the ball was kickable. A qualification agent therefore only shoots to the goal if the distance is not too large and if the scoring angle is wide enough. Furthermore, he does not kick the ball to a random corner in the goal but to the corner which provides him with the widest shooting angle. However, if the distance to the goal is too large then the agent considers a different action. In the first version of our qualification team the agent cleared the ball forward in this case by kicking it into the widest angle between opponents in a forward direction. As a result, the team often succeeded in moving the ball rapidly to the enemy half of the field. This had as a disadvantage however, that it usually led to a loss of possession since the ball could be picked up by an enemy defender. We therefore extended this version by enabling the agents to pass the ball to a free teammate if one was available. They preferred this to clearing in order to try and keep their team in possession of the ball for a longer period of time.

A problem with this version of the team was that the players often passed the ball back and forth to each other and consequently made no progress towards the opponent's half of the field. As a result, the *FC Portugal* defense was never tested. We therefore adapted the decision procedure by only allowing the agents to pass to a free teammate in a forward direction. If such a teammate could not be found, the agent's action depended on his current position. If the agent was located on his own half he cleared the ball forward. Since the ball then usually moved to the beginning of the enemy half, this often gave the team a good starting point for setting up an attack. From there we implemented a simple wing attacking pattern that was effective against the *FC Portugal* defense. The problem that the ball was often picked up by an opponent when cleared further forward was solved by making a slight alteration to the clearing action when the ball was cleared from the center. Instead of considering all forward directions, the agent then cleared the ball into the widest angle between opponents towards the side of the field. Since it was much less crowded in this area, the opponents were usually not able to intercept the ball quickly there. We thus had to make sure that one of our players could reach it first. To this end, we defined a slightly more aggressive formation by making small changes to the ball attraction factors which were used for *De Meer 5*. As a result, our players were attracted more towards the ball which enabled a wing attacker to intercept it before an opponent did. This player would then dribble with the ball along the wing towards the side of the opponent's penalty area where he gave a cross pass to the central attacker. When the central attacker was able to get to the ball he usually had a good chance to score since the wing player had drawn the goalkeeper towards the side. Note that in addition to this attack we also implemented a simple marking scheme to reduce the effectiveness of the *FC Portugal* wing attack. This marking scheme was much improved in future versions however, and will be discussed in more detail in Section 9.5.3.

The complete action selection procedure for when a qualification agent has control of the ball is shown in Algorithm 9.4. Note that we have omitted the action selection for special play modes such as `kick_in` or `free_kick` and only show the procedure for `play_on` mode. During other play modes one of the agents generally moves to a position directly behind the ball (i.e. facing the opponent's goal) after which he selects an action according to the same procedure that he uses in `play_on` situations. We tested our qualification team by repeatedly playing against *FC Portugal 2000* and tuned it to play well against this opponent. Although we lost more than we won against this team, we were able to beat them on several occasions. Our best result was a 4-2 victory and using the logfile for this game we qualified for *RoboCup-2001*. It must be noted however, that the decision procedure described in this section was designed solely for playing well against *FC Portugal 2000* and represented a temporary solution in order to meet the *RoboCup-2001* qualification requirements. As a result, our qualification team lacked a lot of flexibility.

```
if distance to goal is not too large and scoring angle is wide enough then
  kick ball with maximum power to most open corner in opponent goal
else if there is a free teammate in front of me then
  pass ball to most free teammate in forward direction
else if my position is located on own half then
  clear ball forward into widest angle between opponents
else if my position is located to the side of the opponent's penalty area then
  if there is a free teammate inside the opponent's penalty area then
    pass ball to most free teammate inside the opponent's penalty area
  else
    give 'cross pass' to point just in front of the opponent's penalty area
  end if
else if my position is located on the wing of the opponent's half and no opponents are close then
  dribble with ball along the wing towards the side of the opponent's penalty area
else
  clear ball forward into widest angle between opponents towards the side of the field
end if
```

Algorithm 9.4: Action selection for *UvA Trilearn* qualification team when an agent can kick the ball.

9.5.3 UvA Trilearn Team for German Open 2001

After the *RoboCup-2001* qualification deadline had expired, we designed a new setup for the action selection procedure using the knowledge gathered from previous versions. The resulting team was used during the *German Open 2001* in Paderborn (Germany). We had explicitly divided the action selection process into two distinct parts: one in which the agent determines his action mode depending on the current state of the environment (coarse action selection) and one in which the agent generates an appropriate

```
determineActionMode()

if confidence associated with current ball position is too low then
  return ACT_SEARCH_BALL
else if ball is kickable then
  return ACT_KICK_BALL
else if fastest teammate to ball is me then
  return ACT_INTERCEPT
else if shouldIMark() == true then
  return ACT_MARK
else if distance to strategic position > 1 then
  return ACT_GOTO_STRATEGIC_POSITION
else
  return ACT_WATCH_BALL
end if
```

Algorithm 9.5: Method used by *German Open* agents to determine their action mode in a given situation.

action command based on his action mode for the current situation (refined action selection). Algorithm 9.5 shows the method which is used by our *German Open* agents for determining their action mode. This method can be seen to form the basis of the action selection process. Note that this basis is only slightly different from the decision procedure used by *De Meer 5* (see Algorithm 9.3). The main difference between these two teams is not the nature of the action selection policy however, but the way in which this policy is refined to generate a specific action command. Whereas a *De Meer 5* agent only considers a small number of actions in each situation, a *German Open* agent has many more options from which to choose (especially when he has control of the ball). As a result, the *German Open* agent is more flexible.

In our *German Open* team almost all action modes can be chosen based on information which is directly stored in the world model of the agent. The only exception is the decision whether to mark an opponent. This decision is based on a mapping from teammates to opponents which is constructed using the `shouldIMark` method. This method creates a list containing all the opponents that are located in the agent's defensive quarter of the field (i.e. for which the x-coordinate of their position is smaller than $-$`pitch_length`$/4$) and assigns a priority factor to each of these opponents which represents the importance of marking him. This priority factor depends on the distance from the opponent to the penalty spot and on whether the opponent is currently located on the same side of the field as the ball. The priority factor f_i which is assigned to an opponent i is calculated according to the following formula:

$$f_i = \frac{(1 + 2 \cdot \Theta(sign(p_y^t) = sign(q_y^t)))}{||(\texttt{PENALTY\_SPOT\_X}, 0) - \vec{p}_t||} \tag{9.4}$$

where $\vec{p}_t$ and $\vec{q}_t$ respectively denote the current positions of the opponent and the ball and where $\Theta(\beta)$ is a function that returns 1 when the boolean expression β is true and 0 otherwise. An opponent will thus be assigned a higher priority factor if he is located close to the penalty spot and on the same side of the field as the ball. The list of opponents is then sorted based on this factor and for each opponent in the list it is determined which teammate should be his marker. The highest-priority opponent is assigned to the teammate which is closest to him, the second-highest priority opponent to the closest of the remaining teammates, etc. The resulting mapping can be used by the agent to determine whether he should mark an opponent, and if so, which opponent he should mark. Note that if an opponent has indeed been assigned to the agent (i.e. the method returns *true*) then the player variable *OpponentToMark* is set to this opponent.

After determining their action mode for the current situation, the agents use the `generateAction` method to generate an appropriate action command for this mode. This method is shown in Algorithm 9.6. In

```
generateAction()

mode = determineActionMode()
if mode == ACT_SEARCH_BALL then
  action = searchBall()
else if mode == ACT_KICK_BALL then
  action = determineActionWithBall()
else if mode == ACT_INTERCEPT then
  action = intercept()
else if mode == ACT_MARK then
  action = markOpponent(OpponentToMark, MarkDistance, MARK_BALL)
else if mode == ACT_GOTO_STRATEGIC_POSITION then
  action = moveToPos(getStrategicPosition(), PlayerWhenToTurnAngle, PlayerDistanceBack)
else if mode == ACT_WATCH_BALL then
  action = turnBodyToObject(OBJECT_BALL)
end if
action = adjustDashPowerForStamina(action, mode)
send action command to ActHandler
```

Algorithm 9.6: Method used by *German Open* agents to generate an action command

most cases the mapping from action mode to action command is straightforward and amounts to the direct application of one of the player skills which have been described in Chapter 7. The only exception is the action choice when the ball is kickable. This is more complicated since an active agent needs to take various aspects of the current situation into account to determine the best action for the team. In general, the behavior of an agent with the ball should depend on the position of the ball on the field. We have therefore divided the field into different areas (see Figure 9.4) and made the action choice dependent on the area in which the ball is currently located. If an agent has the ball on his own half, for example, he considers different actions than if he is close to the opponent's goal. Algorithm 9.7 shows the decision procedure which is used by the agents of our *German Open* team when they have control of the ball. This procedure contains a different action selection policy for each of the specified areas and is an extension of Algorithm 9.4. As in our qualification team, the main idea is to get the ball away from the defensive area and clear it to one of the sectors on the side of the field. A wing attacker then moves with the ball towards the side of the opponent's penalty area and gives a cross pass to a free teammate in front of the goal. Here a teammate is labeled as 'free' if there are no opponents inside a cone with this teammate as

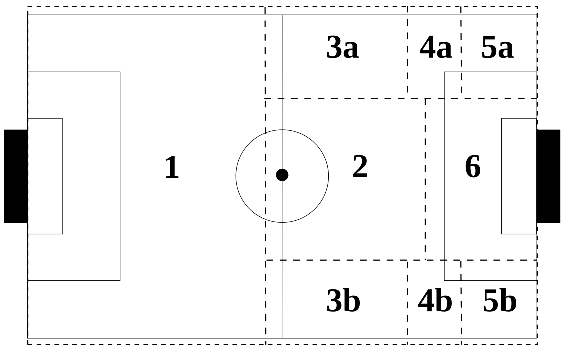


Figure 9.4: Areas on the field which are used for action selection when the ball is kickable.

```
determineActionWithBall()

if distance to ball is not too large and scoring angle is wide enough then
   kick ball with maximum power to most open corner in opponent's goal
else if ball is located in area 1 then
   // consider actions in following order (conditions between brackets):
   turn with ball towards opponent's goal (if facing own goal and no opponents are close)
   pass ball at normal speed to most free teammate in forward direction (only if very free)
   dribble slowly forward in direction of widest angle between opponents (if angle is wide enough)
   clear ball forward into widest angle between opponents (no condition)
else if ball is located in area 2 then
   // consider actions in following order (conditions between brackets):
   clear ball forward to left or right wing depending on widest clearing angle (if angle is wide enough)
   pass ball at normal speed to most free teammate in forward direction (only if very free)
   turn with ball towards opponent's goal (if facing own goal and no opponents are close)
   dribble slowly towards opponent's goal (if no opponents are blocking the desired path)
   clear ball forward into widest angle between opponents (no condition)
else if ball is located in area 3 (i.e. 3a or 3b) then
   // consider actions in following order (conditions between brackets):
   clear ball forward to my side of the field (if teammate − mostly me − can reach ball fastest)
   pass ball at normal speed to most free teammate in forward direction (only if very free)
   pass ball at high speed to most free teammate in forward direction (only if fairly free)
   dribble slowly forward in direction of widest angle between opponents (if angle is wide enough)
   kick ball with maximum power to front edge of opponent's penalty area (no condition)
else if ball is located in area 4 (i.e. 4a or 4b) then
   // consider actions in following order (conditions between brackets):
   dribble fast towards side of opponent's penalty area (if outside penalty area and path not blocked)
   pass ball at normal speed to most free teammate in areas 4, 5 or 6 (only if very free)
   pass ball at high speed to most free teammate in areas 4, 5 or 6 (only if fairly free)
   dribble slowly forward in direction of widest angle between opponents (if angle is wide enough)
   kick ball with maximum power to front edge of opponent's penalty area (no condition)
else if ball is located in area 5 (i.e. 5a or 5b) then
   // consider actions in following order (conditions between brackets):
   pass ball at normal speed to most free teammate in areas 4, 5 or 6 (only if very free)
   pass ball at high speed to most free teammate in areas 4, 5 or 6 (only if fairly free)
   dribble slowly to side of opponent's penalty area (if outside penalty area and path not blocked)
   kick ball with maximum power to front edge of opponent's penalty area (no condition)
else if ball is located in area 6 then
   // consider actions in following order (conditions between brackets):
   pass ball at normal speed to most free teammate in areas 4, 5 or 6 (only if very free)
   pass ball at high speed to most free teammate in areas 4, 5 or 6 (only if fairly free)
   kick ball with maximum power to most open corner in opponent's goal (no condition)
end if
```

Algorithm 9.7: Action selection for *UvA Trilearn German Open* team when an agent can kick the ball.

its base and the ball as its end point. Note that the width of the cone represents a measure of the degree of 'freeness' and is an indication of the required end speed of the pass. If for some reason the desired action cannot be executed (e.g. due to the presence of opponents) several alternatives are considered to make the overall team strategy more flexible. If the desired ball trajectory is blocked by an opponent, for example, the agent might try to dribble with the ball in a different direction even when this means that he must temporarily deviate from the standard attacking pattern. Other options might be to pass the ball to a free teammate if is one is available or to clear the ball forward into the widest angle between opponents. It must be noted however, that despite these alternatives our *German Open* team still lacked the necessary flexibility to play well against different types of opponents.

Algorithm 9.6 shows that the action command which is generated by the agent is passed to a stamina management method called `adjustDashPowerForStamina`. When necessary, this method adapts the command parameters based on the agent's current stamina. Note that this is only relevant if a **dash** command has been generated since stamina is only consumed when the agent dashes. The idea is to determine whether the current situation allows the agent to save his stamina in case it is low. If the agent is currently intercepting the ball or marking an opponent it will be more important for him to keep dashing than if he is moving towards his strategic position. When marking or intercepting, the dash power is therefore only adjusted if the amount of stamina which is lost as a result of the **dash** causes the agent's stamina to drop below the recovery decrement threshold (see Section 3.4.2). This is an undesirable situation since it will lead to a permanent decrease of the agent's recovery value and subsequently to a slower recovery of his stamina during the remainder of the game. To avoid this, the dash power is thus adjusted in such a way that the agent's stamina stays above the threshold. If the agent currently does not have an active part in the game (e.g. because he is located at a large distance to the ball) it is less important for him to keep dashing and this gives him an opportunity to save some stamina. In these cases, we therefore adjust the dash power depending on the difference with the recovery decrement threshold in order to try and keep the agent's stamina well above this threshold. As a result, the agent will have enough stamina to act once he becomes active again. The stamina management procedure is shown in more detail in Algorithm 9.8.

```
adjustDashPowerForStamina(action, mode)

if type(action) ≠ dash then
  return action                                   // only dashing consumes stamina
end if
// backward dash (i.e. dash power < 0) consumes twice as much stamina
sta_loss = ((action→power > 0) ? 1 : -2) · action→power − stamina_inc_max
diff_thr = CurrentAgentStamina − recover_dec_thr · stamina_max
if mode == ACT_INTERCEPT or mode == ACT_MARK then
  if diff_thr < sta_loss then
    action→power = ((action→power > 0) ? 1 : -0.5) · (diff_thr + stamina_inc_max)
  end if
else
  if diff_thr < 0.1 · stamina_max then
    action→power = 0                              // save stamina
  else if diff_thr < 0.25 · stamina_max and distance to ball > 30 then
    action→power = (action→power)/4               // consume less stamina by dashing more slowly
  end if
end if
return action
```

Algorithm 9.8: Method for adjusting the power of a **dash** if an agent's stamina is low.

9.5.4 UvA Trilearn Team for RoboCup-2001

The *UvA Trilearn* team that participated at the *RoboCup-2001* robotic soccer world championship in Seattle (USA) was an extension of our *German Open* team which has been described in Section 9.5.3. In comparison to this team the most important features that were added are the following:

- Heterogeneous players. Team roles are filled by different types of players with different characteristics. Wing attackers, for example, are faster which greatly enhances the effectiveness of the *UvA Trilearn* wing attack. Issues related to heterogeneous player selection were described in Section 9.3.
- Inter-agent communication. The agents use this to increase the reliability of their world state representation. As a result, their ability to cooperate with teammates or mark opponent attackers improves significantly. The *UvA Trilearn* communication model has been described in Section 9.4.
- A scoring policy. This policy enables the agents to determine the best target point in the goal, together with the associated probability of scoring when the ball is shot to this point in a given situation. The underlying statistical framework for this policy was presented in Chapter 8.
- A more refined action selection procedure for when an agent has control of the ball.
- A new goalkeeper that defends his goal line in a more efficient way.

At the *German Open*, our high-level team strategy was too rigid to play well against different types of opponents. Especially defensive opponents proved difficult to beat for us since our agents did not have enough action alternatives during an attack. We therefore extended the action selection strategy when the ball is kickable to increase the flexibility of the team. The general setup for the resulting decision procedure is the same as for the *German Open* agent: the agent first determines his current action mode and subsequently generates an action. If the agent has control of the ball this action depends on the area in which the ball is located (see Figure 9.4). The main difference with respect to the *German Open* agent is that the *UvA Trilearn RoboCup* agent can perform several additional high-level skills such as through passing and outplaying an opponent. When the ball is kickable, he uses a *priority-based* action selection method which is an extension of Algorithm 9.7. Depending on the area in which the ball is located the agent considers various action alternatives which have different priorities (represented by the order in which they are considered). In a given situation he then selects the highest-priority alternative for which the predicted success rate exceeds a certain threshold. Note that we have ordered the alternatives for each area based on our own soccer knowledge and on information contained in soccer literature [14, 38, 54]. In general, the safest and most effective options are tried first and if the predicted success rate for these options is too low then the more risky actions are considered. The exact decision procedure which is used when an agent can kick the ball is shown in Algorithm 9.9. Although with this procedure the agents still have a preference for the standard wing attacking pattern described in Section 9.5.3, their behavior is much more flexible now since they can easily deviate from this pattern and find an alternative way of scoring. As a result, the team is able to perform well against offensive as well as defensive opponents.

For our *UvA Trilearn RoboCup* team we also implemented a new goalkeeper for which the action selection procedure is completely different to that of our old goalie (originating from *De Meer 5*). Instead of moving on a rectangle in front of the goal, the new goalkeeper moves on a line parallel to the goal line at a small distance from the goal. Depending on the position of the ball he picks the optimal guard point on this line as his strategic position. For a detailed explanation of how this optimal guard point is computed we refer the reader to Section 7.4.9. The main feature of our new goalkeeper is that his movement is efficient due to the fact that he keeps the direction of his body aligned with the direction of the line along which he moves. As a result, he is able to move to any point on this line without having to waste cycles on turning

```
determineActionWithBall()

if scoring probability for best scoring point > ScoringProbThr (=0.9) then
  kick ball with maximum power to best scoring point as returned by scoring policy
else if ball is located in area 1 then
  ...     same procedure as for German Open agent; see Algorithm 9.7
else if ball is located in area 2 then
  // consider actions in following order (simplified conditions between brackets):
  turn with ball towards opponent's goal (if facing own goal and no opponents are close)
  dribble fast towards opponent's goal (if no opponents are blocking the desired path)
  pass ball at normal speed to most free teammate in forward direction (only if very free)
  pass ball at high speed to most free teammate in forward direction (only if fairly free)
  give through pass to teammate at the side of the field (if angle between opponents is wide enough)
  clear ball forward into widest angle between opponents (no condition)
else if ball is located in area 3 (i.e. 3a or 3b) then
  // consider actions in following order (simplified conditions between brackets):
  turn with ball towards opponent's goal (if facing own goal and no opponents are close)
  outplay opponent (if opponent is very close)
  dribble fast towards side of opponent's penalty area (if no opponents are blocking the desired path)
  clear ball forward to my side of the field (if teammate − mostly me − can reach ball fastest)
  pass ball at normal speed to most free teammate in forward direction (only if very free)
  dribble slowly forward in direction of widest angle between opponents (if angle is wide enough)
  clear ball forward into widest angle between opponents (no condition)
else if ball is located in area 4 (i.e. 4a or 4b) then
  // consider actions in following order (simplified conditions between brackets):
  dribble fast towards side of opponent's penalty area (if outside penalty area and path not blocked)
  pass ball at normal speed to most free teammate in areas 4, 5 or 6 (only if very free)
  give through pass to teammate in area 6 (if angle between opponents is wide enough)
  outplay opponent (if opponent is very close)
  pass ball at high speed to most free teammate in areas 4, 5 or 6 (only if fairly free)
  clear ball towards penalty area into widest angle between opponents (no condition)
else if ball is located in area 5 (i.e. 5a or 5b) then
  // consider actions in following order (simplified conditions between brackets):
  pass ball at normal speed to most free teammate in areas 4, 5 or 6 (only if very free)
  dribble slowly towards side of opponent's penalty area (if outside penalty area and path not blocked)
  outplay opponent (if opponent is very close)
  give through pass to teammate in area 6 (if angle between opponents is wide enough)
  pass ball at high speed to most free teammate in areas 4, 5 or 6 (only if fairly free)
  clear ball towards penalty area into widest angle between opponents (no condition)
else if ball is located in area 6 then
  // consider actions in following order (simplified conditions between brackets):
  dribble fast towards opponent's goal (if no opponents are blocking the desired path)
  dribble slowly forward in direction of widest angle between opponents (if angle is wide enough)
  pass ball at normal speed to most free teammate in areas 4, 5 or 6 (only if very free)
  give through pass to teammate in area 6 (if angle between opponents is wide enough)
  pass ball at high speed to most free teammate in areas 4, 5 or 6 (only if fairly free)
  kick ball with maximum power to best scoring point as returned by scoring policy (no condition)
end if
```

Algorithm 9.9: Action selection for the *UvA Trilearn RoboCup* team when an agent can kick the ball.

his body. Throughout the match, the goalkeeper will stay on his line for as long as possible. When he is in a passive situation (i.e. the ball is far away or not heading towards the goal) he uses the `defendGoalLine` skill to move to the optimal guard point on the line. Furthermore, he will even stay on his line when the ball is heading for the goal and will enter the goal in less than 20 cycles. In this case, he moves to the intersection point of the ball trajectory with the line in an attempt to block the path to the goal and catch the ball once it comes within his catchable distance. In general, it will take the goalkeeper fewer cycles to catch the ball in this manner (i.e. without having to turn) than if he would use the `intercept` skill. The only situation in which the goalkeeper leaves his line is when he is clearly the fastest player to the ball and when the point of interception is inside his own penalty area. The complete decision procedure for the *UvA Trilearn* goalkeeper is shown in Algorithm 9.10.

if confidence associated with current ball position is too low **then**
  return `searchBall()`
else if ball is catchable **then**
  return `catchBall()`
else if fastest player to ball is clearly me and interception point lies inside penalty area **then**
  return `intercept()`
else if ball is heading towards goal and will enter goal in $<$ 20 cycles **then**
  // $\vec{z}$ = desired guard point, $\theta^t + \phi^t$ = agent's global body angle in cycle t
  $\vec{z}$ = intersection(ball trajectory, goalie line)
  return `moveToPosAlongLine`($\vec{z}$, $\theta^t + \phi^t$, *GoalieDevDist*, 1, *GoalieTurnThr*, *GoalieCorrAng*)
else
  return `defendGoalLine`(*DefendGoalLineDist*)
end if

Algorithm 9.10: Action selection procedure used by the *UvA Trilearn* goalkeeper.

9.6 Results

Although the features that have been described in this section have all been merged into a single robotic soccer system, we can isolate the effects of each of them through controlled testing. A proper way to establish the effectiveness of a technique x that is a single aspect of a team is to let this team play games against itself in which one side uses technique x and the other side does not. In this section we present empirical results that demonstrate the effectiveness of our use of heterogeneous players, our communication model and our new goalkeeper. Although a full-length game in the *soccer server* lasts for 6,000 cycles (10 minutes), the game scores generally vary greatly due to the large amount of noise which is incorporated into the simulation. The reported results are therefore based on several games. Compiled statistics include the number of games won, drawn and lost by a team, the average number of goals scored by a team and the standard deviation from this average. Finally, Section 9.6.4 presents results of games between the teams described in Sections 9.5.1−9.5.4 and the top three teams at *RoboCup-2000*. These results are meant to give a rough estimate of the overall strength of *UvA Trilearn* and its intermediate versions. All the tests described in this section have been performed using three Pentium III 1GHz/256MB machines (one for the server and one for each team) running Red Hat Linux 7.1.

9.6.1 Heterogeneous Player Results

As described in Section 9.3, the *UvA Trilearn* team uses faster players at the wings of the formation to enhance the effectiveness of their wing attack. We have tested the benefit this approach by playing matches between our standard *UvA Trilearn* team (that uses heterogeneous players) and a homogeneous

UvA Trilearn team that uses only default players. The behavior of the players on both teams was otherwise identical. Table 9.3 shows the results over the course of 10 full-length games. Here it must be noted that the outcome of a match cannot be regarded as a measure for the actual performance of a heterogeneous player. However, it does give a good indication of the overall effect of using them. The results clearly speak in favor of the heterogeneous team which did not lose a single game and did not concede a single goal. This superiority was also visible during the games. The heterogeneous players were able to move faster when they intercepted the ball and easily outran their opponents on an attack[6]. The homogeneous team, on the other hand, hardly ever managed to create a scoring opportunity due to their lack of speed which gave the heterogeneous team enough time to reorganize their defense (which also contained faster players on the wings). As a result, the wing attack of the homogeneous team was less effective. To test the statistical significance of the results we also played the heterogeneous team against itself and the homogeneous team against itself. Over the course of 10 games, the heterogeneous team scored an average of 1.4 goals with a standard deviation of 1.3 and the homogeneous team scored an average of 0.3 goals with a standard deviation of 0.48. These outcomes confirmed our earlier observations: the heterogeneous team manages to score about 1.4 goals per game against the *UvA Trilearn* goalkeeper, whereas the homogeneous team fails to score often due to their lack of speed on the wings which reduces the effectiveness of the team strategy.

	Wins	**Draws**	**Losses**	**Av. score**	**St. dev.**
Heterogeneous players	7	3	0	1.4	1.67
Homogeneous players	0	3	7	0.0	0.00

Table 9.3: Results of 10 games between a homogeneous and a heterogeneous *UvA Trilearn* team.

9.6.2 Communication Results

The *UvA Trilearn* agents use communication to increase the reliability of their world model (see Section 9.4). Depending on the position of the ball the agent that has the best view of the field (either the left or right midfielder in our implementation) communicates his world model to all nearby teammates. We have tested the effect of our communication model on the overall team performance by playing matches between our standard *UvA Trilearn* team (that makes use of communication) and an adapted version of this team that uses no communication. Besides the ability to communicate, the behavior of the players on both teams was identical. Table 9.4 shows the results over the course of 10 full-length games. These results clearly show that communication has a positive effect on the performance of the team. The team that makes use of communication wins almost every game and concedes only three goals in 10 matches. Furthermore, the world model of the communicating agents contains up-to-date information about a larger number of players on the field. On average, the communicating agents 'see' about 17 players in each cycle, whereas this number is only about 13 for the non-communicating agents. Despite the fact that each agent only has a limited view cone from which he receives visual information, the communicating agents thus still have a good global representation of the environment. As a result, the communicating agents have more up-to-date player information to consider during the action selection process and this increases their ability to cooperate with other teammates. To test the statistical significance of the results, we again

	Wins	**Draws**	**Losses**	**Av. score**	**St. dev.**	**Players seen**	**St. dev.**
Communication	8	2	0	1.8	1.13	16.793	0.252
No communic.	0	2	8	0.3	0.48	12.807	0.180

Table 9.4: Results of 10 games between *UvA Trilearn* with and without communication.

[6]Despite this, the number of goals scored was rather low. This was mainly due to the goalkeeper that made a lot of saves.

played the non-communicating team against itself. Over the course of 10 games, this team scored an average number of 0.6 goals with a standard deviation of 0.84. The results of the standard team (which uses communication) against itself were already given in Section 9.6.1: over 10 games they scored an average of 1.4 goals with a standard deviation of 1.3. When compared to Table 9.4, these results are as can be expected: the communicating team scores slightly less often against itself (better opponent), whereas the non-communicating team scores slightly more often (worse opponent).

9.6.3 Goalkeeper Results

To test whether our new goalkeeper (which moves on a line; see Section 9.5.4) is an improvement over the old one (which moves on a rectangle; see Section 9.5.1) we played 10 full-length games between *UvA Trilearn* with the old goalkeeper and *UvA Trilearn* with the new goalkeeper. Besides the goalkeeper both teams were identical. Table 9.5 shows the results. These results clearly show that the introduction of the new goalkeeper had a positive effect: the team that uses the new goalkeeper wins all of its games and concedes only very few goals. When we played the team with the old goalkeeper against itself, an average number of 4.2 goals were scored by each team (over the course of 10 games) with a standard deviation of 1.69. This shows that in general it is much easier to pass the old goalkeeper than the new one.

	Wins	**Draws**	**Losses**	**Av. Score**	**St. Dev.**
New goalkeeper	10	0	0	4.8	1.81
Old goalkeeper	0	0	10	1.0	0.67

Table 9.5: Results of 10 games between *UvA Trilearn* with the old and the new goalkeeper.

9.6.4 Overall Team Results

To get an indication of the performance of the *UvA Trilearn* team and its intermediate versions relative to the best teams from the previous year, we played a trial competition that contained the teams described in Sections 9.5.1−9.5.4 and the top three teams at *RoboCup-2000*. In this competition all the teams played 10 games against each other. The results are summarized in Tables 9.6 and 9.7. Table 9.6 gives an indication of the overall strength of the teams relative to each other. It contains a matrix of all the matches that were played and shows the number games won, drawn and lost by a team against each of the other teams. The last column denotes the number of points that were gathered by each team in all its matches according to the standard soccer rules: three points for a win and one for a draw. This points measure can be seen to create an ordering between the teams indicating their strength in the competition. Table 9.7 shows the cumulative score over the course of 10 games between each of the teams. This gives an indication of which teams are able to score easily or concede fewer goals against specific other teams.

Before we explain the results several remarks are in order. First of all, the matches were played using *soccer server* version 7.10 while the *RoboCup-2000* teams were originally built for version 6.xx. The main differences between versions 6 and 7 of the server are (1) in version 7 the ball can be accelerated more with a single kick, and (2) in version 7 the stamina of the players increases more in each cycle. Although the players of each team were aware of the changed settings (which are read from a configuration file before the start of a match), it was clear that the *RoboCup-2000* teams did not adapt their strategy accordingly. The *FC Portugal 2000* team, for example, is normally highly configurable and uses a different strategy for different types of opponents by changing the team's configuration. This is done manually before the game and automatically during the game when the situation asks for it. However, the publicly released *FC Portugal 2000* binary that was used in our experiments was created specifically to play well with a lower stamina increase per cycle. As a result, the team was not as aggressive as they could have been with the new

Team	1	2	3	4	5	6	7	Points
1. De Meer 5	-	6-2-2	2-2-6	0-0-10	1-2-7	0-0-10	3-4-3	46
2. Trilearn Qualification	2-2-6	-	0-2-8	0-0-10	4-0-6	1-2-7	4-4-2	43
3. Trilearn German Open	6-2-2	8-2-0	-	0-0-10	4-0-6	2-0-8	5-4-1	83
4. Trilearn RoboCup	10-0-0	10-0-0	10-0-0	-	10-0-0	10-0-0	10-0-0	180
5. FC Portugal 2000	7-2-1	6-0-4	8-0-2	0-0-10	-	6-3-1	10-0-0	116
6. Brainstormers 2000	10-0-0	7-2-1	8-0-2	0-0-10	1-3-6	-	10-0-0	113
7. ATT-CMU 2000	3-4-3	2-4-4	1-4-5	0-0-10	0-0-10	0-0-10	-	30

Table 9.6: Results of matches played between four versions of the *UvA Trilearn* team and the top three teams at *RoboCup-2000*. Each entry shows the number of games won, drawn and lost by a team against a specific opponent. The right-most column denotes the number of points that a team has gathered (three for a win and one for a draw) and can be used to create an ordering of the teams indicating their strength.

Team	1	2	3	4	5	6	7	Total
1. De Meer 5	-	22-6	5-9	0-96	5-21	0-15	4-7	36-154
2. Trilearn Qualification	6-22	-	10-25	0-110	20-28	3-12	10-6	49-203
3. Trilearn German Open	9-5	25-10	-	0-107	44-40	4-19	11-7	93-188
4. Trilearn RoboCup	96-0	110-0	107-0	-	180-0	34-0	78-0	605-0
5. FC Portugal 2000	21-5	28-20	40-44	0-180	-	11-6	41-3	141-258
6. Brainstormers 2000	15-0	12-3	19-4	0-34	6-11	-	44-0	96-52
7. ATT-CMU 2000	7-4	6-10	7-11	0-78	3-41	0-44	-	23-188

Table 9.7: Cumulative scores of matches between four versions of *UvA Trilearn* and the top three teams at *RoboCup-2000*. Each entry denotes the number of goals scored for and against a team when playing a specific opponent over the course of 10 games. The right-most column shows the total score for the teams.

server settings. Our results against this team are therefore slightly flattered. Furthermore, the complexity of the simulation causes the result of a game to be influenced by many different factors. Consequently, there is a potential danger of drawing invalid conclusions from results such as those presented in Tables 9.6 and 9.7. Some examples are the following:

- It is invalid to conclude that if team A beats team B then all of the techniques used by team A are more successful than those used by team B. Unless both teams are identical except in one respect, no individual aspect of either team can conclusively be credited with or blamed for the result.
- The final score of a game is no measure for the abilities of the teams that played in it. It cannot be derived from the result, for example, which team used the best positioning method, which team had the best goalkeeper, which team possessed the best individual skills, etc. The result of a game only gives an indication of the combined effect of all the team's characteristics and as such it can generally be seen as a test of the overall team strength and not of the individual components.
- Game results are not 'transitive'. It cannot be concluded that if team A beats team B and team B beats team C then team A will beat team C. This is because different teams use different strategies which might be more successful against certain opponents and less successful against others. The *UvA Trilearn* qualification team, for example, is tuned to play well against *FC Portugal 2000* and is able to beat this team on several occassions. However, *FC Portugal 2000* wins most of its matches against *Brainstormers 2000*, whereas our qualification team nearly always loses against them.

- The goal difference during a game is no measure for how much better the winning team has performed overall as compared to the losing team. The score can be flattered due to the fact that the losing team unsuccessfully changed its strategy during the match in an attempt to force a positive result. The *FC Portugal 2000* team, for example, changes their formation and starts to play very offensively if they are trailing by two or more goals. However, in most cases the result is counterproductive to the intention since their defense will become weaker which makes it easier for the opponents to score more often. As a result, the team is usually beaten heavily once the opponents lead by two goals.

When we look at the results of the top three teams at *RoboCup-2000*, it is clear that *FC Portugal* is an offensive team: they manage to score a lot but also concede quite a large number of goals[7]. *Brainstormers*, on the other hand, are a defensive team: they conceded very few goals but were not able to score a lot either. Finally, *ATT-CMU* was not able to score often and also conceded many goals. The results of the incremental versions of *UvA Trilearn* show that in general each version was a clear improvement over the previous one. The only exception was the *UvA Trilearn* qualification team that overall gathered fewer points than *De Meer 5*. However, the purpose of our qualification team was to play well against *FC Portugal 2000* in order to qualify for *RoboCup-2001* and the results clearly show that our qualification team performs better against *FC Portugal 2000* than *De Meer 5* does. The high-level strategy of *De Meer 5* was too simple to score many goals against any opponent. However, they also conceded relatively few goals due to the fact that they always kicked the ball to the opponent's half of the field which made it difficult for the opponent team to set up an attack. The qualification team was able to score more goals since the agents were now able to pass the ball around. As a result, they could keep the attack going. This had as a disadvantage however, that the ball was lost on the defensive half more often (when the opponents intercepted a pass) than would happen to *De Meer 5* and this explains the higher amount of goals conceded. The *UvA Trilearn German Open* team is a clear improvement over the earlier versions and performs reasonably well against most teams. *Brainstormers* proved to be the most difficult opponent for this team due to their tight defense which made it difficult to score. Despite this, our *German Open* team succeeded in beating them on two occasions. Finally, our *UvA Trilearn RoboCup* team clearly outperformed every other team that they played against. This team won all its matches and was able to score 605 goals in 60 games without conceding a single goal. As compared to the *German Open* team, our *RoboCup* team was much more flexible and could play well against offensive opponents (*FC Portugal*) as well as defensive opponents (*Brainstormers*). Although the tight *Brainstormers* defense still proved tough to penetrate, our *RoboCup* team managed to score three to four goals in each game against them.

9.7 Conclusion

In this chapter we have described the high-level team strategy of the *UvA Trilearn* soccer simulation team that participated at the *RoboCup-2001* robotic soccer world championship. In addition, we have also presented three intermediate versions of this team which were respectively used for testing our low-level implementation, to qualify for *RoboCup-2001* and to compete at the *German Open 2001*. These intermediate versions have shown that the development of the *UvA Trilearn* team strategy has been an incremental process. We have demonstrated the effectiveness of the individual components of this strategy (heterogeneous players, communication, etc.) and showed that the combination of these features enables the team to play well against different types of opponents. The final version of the *UvA Trilearn* team that was described in this chapter was able to comprehensively beat the top three teams from the previous world championship. Since these three teams were the best teams that we could practice against before the start of the next world cup, this concluded our preparation for *RoboCup-2001*.

[7] It must be noted that the number of goals scored against *FC Portugal* is flattered in these results due to the fact that they change their formation when trailing by two goals. Furthermore, the incremental versions of *UvA Trilearn* have all been extensively tested against this team.

Chapter 10

Competition Results

During the project, the *UvA Trilearn 2001* soccer simulation team has participated in two international robotic soccer competitions: the *German Open 2001* and the official *RoboCup-2001* world championship. In this chapter we present our results in these competitions and discuss our performances. When reading this chapter it is important to realize that robotic soccer competitions cannot be regarded as controlled experiments. We therefore want to emphasize the fact that we do not present our competition results as a scientific validation of the techniques that we have used. These techniques have been discussed in the previous chapters along with a number of controlled experiments which serve as their empirical validation. However, we do believe that competition results can be seen as a useful evaluation of the system as a whole. Participating in robotic soccer competitions has given us information concerning the strengths and weaknesses of the various approaches that we used. As such, they have been of critical importance for the development of our team. This chapter is organized as follows. In Section 10.1 we discuss several advantages and disadvantages of robotic soccer competitions and present an overview of past competition results. In Section 10.2 we then discuss the results of *UvA Trilearn* at the *German Open 2001*, while Section 10.3 will describe our performance at the *RoboCup-2001* world championship in Seattle (USA).

10.1 Introduction

Although robotic soccer as a discipline clearly has a very competitive nature, it is important to realize that RoboCup is primarily a research initiative. The presence of competitions is therefore no prerequisite for the domain to exist. However, organizing robotic soccer competitions has several advantages. For example, competitions form hard deadlines for creating complete working systems. If a team wants to compete they need to get all the system components operational and working together. The natural desire to perform well can then provide a strong motivation for solving the challenging aspects of the domain. Furthermore, robotic soccer competitions bring researchers together who have all tried to solve the same problems. As such, the competitions provide a common platform for exchanging ideas. Since all the participants have implemented their ideas in the same underlying architecture it is relatively easy to compare different approaches using this standard test bed. An additional benefit is that competitions cause a wide pool of teams to be created. After each competition a large number of source codes and binary files from participating teams are made publicly available and can be used for controlled testing of various techniques by others. This and the fact that many papers are published containing the research contributions of these teams lead to continually improving solutions in each competition, since all the participants know that in order to win they must be able to outperform the top teams from the previous event[1]. It can thus be concluded that competitions have the potential to accelerate scientific progress within the robotic soccer domain.

However, organizing robotic soccer competitions also involves a number of potential dangers which might slow down scientific progress. The most obvious one is that winning competitions becomes the main priority at the expense of all else, including science. Especially if monetary prizes are awarded, many people will focus only on winning and are likely to keep successful techniques secret from each competition to the next. Therefore, monetary prizes are generally not given. In order to keep the focus on scientific contributions, 'scientific challenge' awards are presented to teams who have demonstrated the best scientific research results regardless of their performance in competitions. Another potential danger is that winning solutions will be tuned too much to the low-level details of the domain. If the competitions are to serve science however, the winning techniques should obviously be generally applicable beyond the domain in question. Although in general it will not be possible to avoid some domain-dependent solutions, the competitions should be such that these cannot be sufficient to produce a winning team.

Pre-RoboCup-96 was the first simulated robotic soccer competition using *soccer server*. It was held in Osaka (Japan) in conjunction with the IROS-96 conference and was meant as an informal competition to test the server in preparation for *RoboCup-97*. The teams in this tournament generally used very simple strategies keeping their players in fixed locations and only moving them towards the ball when it was close. *RoboCup-97* was held in Nagoya (Japan) in conjunction with the IJCAI-97 conference and was the first formal simulated robotic soccer competition. This competition was won by the team that exhibited the best low-level skills. As the years progressed however, the crucial differences between the teams were found more towards the higher strategic levels. The reason for this was that the low-level techniques that had proven to be successful were used as a basis by other teams in subsequent competitions. As a result, the differences among participating teams shifted to the more general levels. Interesting in this respect was that as of 1998 it became a tradition that the previous champion participated with minimal modifications in order to measure progress from each year to the next. At *RoboCup-98* in Paris (held in conjunction with the ICMAS-98 conference) the *RoboCup-97* champion *AT Humboldt* participated as the benchmark team and finished roughly in the middle of the final standings. This proved that the field of entries as a whole was much stronger than the year before. In subsequent competitions however, the benchmark team started to perform better indicating that it became more difficult to outperform the previous champion. This was a direct result of the fact that most teams had solved their low-level problems and shifted their focus towards the strategic levels. The best performance of a benchmark team came at *RoboCup-2000* in Melbourne (held in conjunction with the PRICAI-2000 conference) where the 1999 champion *CMUnited* reached fourth place. Apart from the official world championships which have been held each year since 1996, several other competitions have also been organized. The top three teams of all the official RoboCup competitions that have taken place (including those in 2001) are shown in Table 10.1.

In the remainder of this chapter we discuss the results of the *UvA Trilearn 2001* soccer simulation team at the *German Open 2001* and at the *RoboCup-2001* world championship. Although robotic soccer competitions cannot be regarded as controlled experiments, we do believe that competition results can be seen as a useful evaluation of the system as a whole. Since the overall goal is to create a team of agents that can operate in an adversarial environment, it is interesting to observe how the team performs against a wide range of previously unseen opponents. As such, participating in robotic soccer competitions provides an insight into the strengths and weaknesses of various approaches.

[1] Note that this benefit only holds if similar rules are used as the basis for competition each year.

Competition	**Location**	**Winner**	**Runner-up**	**Third**	**Nr**
Pre-RoboCup-1996	Osaka	Ogalets	Sekine	Waseda	8
RoboCup-1997	**Nagoya**	**AT Humboldt**	**Andhill**	**ISIS**	**29**
Japan Open 1998	Tokyo	Andhill	Kasuga-bito II	NITStones	10
Pacific Rim 1998	Singapore	Kasuga-bito II	KU-Sakura	Cyberoos	9
RoboCup-1998	**Paris**	**CMUnited**	**AT Humboldt**	**Windmill W.**	**34**
Japan Open 1999	Nagoya	11 monkeys	YowAI	Gullwing	11
RoboCup-1999	**Stockholm**	**CMUnited**	**Magma Freiburg**	**Essex Wizards**	**37**
Japan Open 2000	Hakodate	YowAI	11 Monkeys 2	TakAI	24
China RC 2000	HeFei City	Tsinghuaeolus	USTC II	USTC I	10
EuRoboCup-2000	Amsterdam	FC Portugal	Brainstormers	Essex Wizards	13
RoboCup-2000	**Melbourne**	**FC Portugal**	**Brainstormers**	**ATTCMUnited**	**34**
German Open 2001	Paderborn	FC Portugal	Brainstormers	Sharif Arvand	12
Japan Open 2001	Fukuoka	YowAI	FC Tripletta	Team Harmony	27
RoboCup-2001	**Seattle**	**Tsinghuaeolus**	**Brainstormers**	**FC Portugal**	**42**
China RC 2001	Kunming	Wright Eagle	Shu 2001	EveRest	12

Table 10.1: The top three teams of all the official RoboCup competitions that have taken place. The last column denotes the number of participants (excluding non-qualified teams and withdrawals).

10.2 German Open 2001

The *German Open 2001* was held at the Heinz Nixdorf Museums Forum in Paderborn (Germany) from the 8th-10th June, 2001. It was the first official robotic soccer competition in which *UvA Trilearn 2001* participated and mainly served as a testcase for the *RoboCup-2001* world championship later that year. Because of our initial decision to build a team from scratch, the majority of our time up to that point had been spent on solving low-level problems. As a result, *UvA Trilearn* used a very simple and rigid high-level strategy during this tournament in which each agent only had a few high-level skills to choose from. A total number of 16 teams from six different countries had registered for the competition, four of which withdrew before the start. The 12 remaining teams were divided into two groups of six and played a round-robin tournament against the teams in their group. The first four teams from each group proceeded to the next stage that was played according to a double elimination system[2]. This meant that the eight remaining teams could not be eliminated from the competition before losing at least twice. After losing one match a team thus still had a chance to win the tournament and this increased the likelihood that the final standings would actually reflect the true order of merit between the teams.

Table 10.2 shows the results and several statistics of all the games played by *UvA Trilearn* during the competition. In the round-robin stage we played five matches of which two were won, two were lost and one was drawn. Two results were significant: the 2-5 loss against *FC Portugal* and the 16-0 win against *Osna BallByters*. The former meant that *UvA Trilearn* became the first ever team in an official competition that managed to score two goals in one match against *FC Portugal*, who had won both *EuRoboCup-2000* and *RoboCup-2000* without conceding a single goal. The match against *Osna BallByters* was important because *UvA Trilearn* had to win by a difference of 15 goals in order to avoid finishing fourth in the group

[2] In this format each remaining team starts out in the winner's bracket and continues to compete for the title until it has lost two matches. When a team loses for the first time it is moved to the loser's bracket where it plays against other teams that have lost once. Eventually, the winner of the loser's bracket will play the winner of the winner's bracket in the final. If this final is won by the team that won the winner's bracket this team is the champion. If the loser's bracket winner wins the final however, both teams have lost once and have to play again to determine who becomes the champion.

Round	Opponent	Affiliation	Score	Poss	Def	Mid	Att
Group	*Aras*	Sharif Univ. of Technology	2-2	46%	13%	66%	21%
	FC Portugal	Universities of Aveiro/Porto	2-5	44%	24%	54%	22%
	Lucky Lübeck	University of Lübeck	4-0	70%	17%	56%	27%
	Sharif Arvand	Sharif Univ. of Technology	1-2	59%	34%	43%	23%
	Osna BallByters	University of Osnabrück	16-0	60%	0%	30%	70%
Elimination	*Dr. Web*	Saint-Petersburg University	0-1	48%	17%	48%	35%
	RoboLog	University of Koblenz-Landau	4-0	57%	17%	49%	34%
	Dr. Web	Saint-Petersburg University	0-1	49%	9%	50%	41%
			29-11	54%	16%	50%	34%

Table 10.2: Results and several statistics of all the games played by *UvA Trilearn* at the *German Open 2001*. The 'Poss' column denotes the percentage of the total time during a match in which *UvA Trilearn* was in ball possession. The last three columns respectively denote the percentage of the time in which the ball was located in our defensive, middle and attacking part of the field. Note that these zones correspond to three equal-length field parts over the full width of the field. The statistics were generated by *RoboBase*, a logplayer and analysis tool for RoboCup logfiles [87]. *UvA Trilearn* reached 5th place in the competition.

and having to play the winner of the other group in the next stage. In an exciting goal chase we succeeded in beating *Osna BallByters* by 16-0 which meant that we finished the group in third place. In the winner's bracket of the double elimination stage we lost our first match by 0-1 against the defensive *Dr. Web* who were rarely able to reach our defensive area but nevertheless managed to score once. In the loser's bracket we then beat the Prolog-based German team *RoboLog* after which we again lost a very close match by 0-1 against *Dr. Web*[3] in sudden-death overtime. This resulted in a fifth place in the final standings which are shown in Table 10.3. The tournament was won by *FC Portugal* who beat *Karlsruhe Brainstormers* in a repeat of the *RoboCup-2000* final.

Place	Team	Affiliation
1.	*FC Portugal*	Universities of Aveiro/Porto, Portugal
2.	*Brainstormers*	University of Karlsruhe, Germany
3.	*Sharif Arvand*	Sharif University of Technology, Iran
4.	*Dr. Web*	Saint-Petersburg University, Russia
5.	***UvA Trilearn***	**University of Amsterdam, The Netherlands**
5.	*Aras*	Sharif University of Technology, Iran
7.	*RoboLog*	University of Koblenz-Landau, Germany
7.	*Rolling Brains*	University of Mainz, Germany

Table 10.3: Final standings (top eight teams) of the RoboCup *German Open 2001*.

During the tournament we discovered that the quality of the participating teams was much higher than the year before. The differences between the teams were smaller and especially the group into which *UvA Trilearn* was drawn proved to be very strong (the four teams that qualified from this group all finished in the top six of the final standings). A lot of teams used a formation mechanism similar to *Situation Based Strategic Positioning* (SBSP) introduced by *FC Portugal* a year before [56, 77] and this made the positioning of players much more controlled. The most important conclusion that could be drawn from

[3] It was actually a mistake of the organization that we had to play *Dr. Web* so soon again, since according to the official double elimination rules *Dr. Web* should have been placed in another part of the loser's bracket.

the competition was that our high-level strategy was not flexible enough to play well against different types of opponents. Especially teams that played defensively proved difficult to beat for us since we had trouble creating scoring opportunities against them. This was visible, for example, in the matches that we played against *Dr. Web* where we did not succeed in scoring a goal despite the fact that most of the time the action was located in their defensive area. It was clear that we had practiced too much against *FC Portugal 2000* and that our high-level strategy was tuned to perform well against teams that used the same offensive playing style. Nevertheless, the results at the *German Open* were promising since they convinced us that our lower levels worked well and that even with our rigid high-level strategy we were competitive. As such, the tournament had been a good test case for *RoboCup-2001* two months later.

10.3 The RoboCup-2001 World Championship

The *RoboCup-2001* robotic soccer world championship was held at the Washington State Convention Center in Seattle (USA) from the 2nd-10th August 2001 in conjunction with the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI-01). *UvA Trilearn* had qualified for this tournament by means of a qualification logfile [47] in which we beat the *RoboCup-2000* champion *FC Portugal 2000* by 4-2 and by a two-page research abstract describing the research focus and scientific contributions of the team [20]. In comparison to the *German Open* we had improved our team in several areas by introducing inter-agent communication, heterogeneous players and an advanced scoring policy. Furthermore, we improved the team's high-level strategy and increased its flexibility against different types of opponents. This was done by extending the decision procedure of the players thereby giving them more options for choosing an action in different situations. In addition, we improved our defensive strategy and implemented a new goalkeeper with better interceptive capabilities than the one we used at the *German Open*. A total number of 86 teams from 20 different countries had pre-registered for *RoboCup-2001* of which 45 teams qualified for the main event. Since three teams withdrew just before the start of the competition, the tournament consisted of 42 teams which were divided into eight groups (six of five teams and two of six). The first group stage was played according to a round-robin system in which the first three teams qualified for the next round. After this, the 24 remaining teams were divided into four groups of six which again played a round-robin from which the first two teams progressed to an eight-team double elimination stage.

Table 10.4 shows the results and several statistics of all the games played by *UvA Trilearn* during the competition. The tournament started off dramatically for us with a 0-7 defeat in the very first game against the eventual winner *Tsinghuaeolus* from China. The main problem that we had with this team was that their players were very good at controlling the ball at high speed and keeping it away from close opponents. For this they used a dribble-like action in which they moved the ball quickly from side to side within their kickable range. As a result, our defenders were often outplayed since they predicted that the ball would move very fast to the other side of the field due to its speed[4]. Nevertheless, *UvA Trilearn* recovered well finishing second in the group after winning the remaining group games without conceding a single goal. Especially our 6-0 victory against *Living Systems*, the successor team to *RoboCup-99* runner-up *Magma Freiburg*, was a promising result at this stage. The second group stage was even more successful for us: *UvA Trilearn* became first in the group after winning all five matches without conceding a single goal. This included an 8-0 victory against first round group winner *Cyberoos* from Australia. However, the most significant result in this round came against *RoboCup-2000* winner *FC Portugal 2000* (the benchmark team; see Section 10.1) who had won all their matches in the first group stage by a

[4]After the first group stage we corrected this by setting the estimated ball speed to zero when a player has the ball within his kickable range. As a result, our defenders exhibited a more stable behavior and were not so easily outplayed. A number of practice matches between *UvA Trilearn* and *Tsinghuaeolus* that were played after the competition actually showed that with this modification the difference between both teams was much smaller: on average *UvA Trilearn* lost by about 1-3.

Round	Opponent	Affiliation	Score	Poss	Def	Mid	Att
Group I	*Tsinghuaeolus*	Tsinghua University	0-7	37%	44%	45%	11%
	Living Systems	Living Systems AG	6-0	67%	11%	53%	36%
	Harmony	Hokkaido University	3-0	54%	25%	40%	35%
	rUNSWift	Univ. of New South Wales	2-0	60%	20%	42%	38%
Group II	*TUT-groove*	Toyohashi Univ. of Techn.	6-0	56%	8%	52%	40%
	A-Team	Tokyo Institute of Techn.	4-0	62%	6%	59%	35%
	FC Portugal 2000	Univ.'s of Aveiro/Porto	6-0	59%	27%	49%	24%
	Cyberoos	CSIRO (Australia)	8-0	53%	12%	49%	39%
	Rolling Brains	University of Mainz	5-0	53%	2%	57%	41%
Elimination	*Wright Eagle*	Univ. of Science/Techn.	2-1	50%	19%	49%	32%
	FC Portugal 2001	Univ.'s of Aveiro/Porto	1-4	49%	42%	42%	16%
	FC Portugal 2000	Univ.'s of Aveiro/Porto	6-1	60%	29%	59%	12%
	Brainstormers	University of Karlsruhe	0-1	45%	35%	31%	34%
			49-14	54%	22%	48%	30%

Table 10.4: Results and several statistics of all the games played by *UvA Trilearn* at *RoboCup-2001*. The statistics were generated by *RoboBase*, a logplayer and analysis tool for RoboCup logfiles [87]. For the meaning of these statistics we refer to Table 10.2. *UvA Trilearn* reached 4th place in the competition.

combined score of 82-0. We beat *FC Portugal 2000* by 6-0 and this made us the first team ever to record a victory against them in an official competition. In the winner's bracket of the double elimination stage we won a tight first match against the Chinese team *Wright Eagle* by 2-1 in sudden-death overtime after a mistake of their goalkeeper. In the semi-final we then played the tournament favourites *FC Portugal 2001*, the successor team to *RoboCup-2000* champion *FC Portugal 2000*. We lost this game by 1-4 although we were still leading by 1-0 after three-quarters of the total playing time. At this point however, *FC Portugal 2001* changed their formation by moving their goalkeeper to the center line which gave them an extra field player. This new strategy proved to be very effective. We were no longer able to penetrate their defense (which was positioned very far forward now) and the resulting pressure enabled them to score four goals at the end of the match. In our first match in the loser's bracket we again had to play against the benchmark team *FC Portugal 2000* whom we beat by 6-1 on this occasion. We were then eliminated from the competition by *RoboCup-2000* runner-up *Karlsruhe Brainstormers* who beat us by 0-1 in a close match. The *Brainstormers* took the lead with an early goal and although we managed to put pressure on our opponents on several occasions we were not able to score due to their tight defense and good goalkeeper. Our best chance to equalize came towards the end of the match when a shot just missed the goal. This meant that *UvA Trilearn* finished the competition in fourth place. The final of *RoboCup-2001* was between *Tsinghuaeolus* and *Karlsruhe Brainstormers* who had beaten *FC Portugal 2001* in the loser's bracket final. *Tsinghuaeolus* won the match by 1-0 in sudden-death overtime.

The final standings (top eight teams) of the *RoboCup-2001 Simulation League* tournament are shown in Table 10.5. During the competition it was clear that the *UvA Trilearn* high-level strategy had become more flexible as compared to the *German Open*. We were now able to play well against different types of opponents and also managed to score goals against defensive teams. The main philosophy of *UvA Trilearn* was to keep the ball moving quickly from each player to the next and preferably in a forward direction. In addition, the players often tried to cut through the opponent's defense by passing the ball into the depth in front of the wing attackers at the side of the field. In this way, the team often succeeded in moving the ball rapidly to the enemy half of the field thereby disorganizing the opponent team. The effectiveness of this strategy was greatly enhanced by the fact that we used heterogeneous players on the wings. Although

Place	Team	Affiliation
1.	*Tsinghuaeolus*	Tsinghua University, China
2.	*Karlsruhe Brainstormers*	University of Karlsruhe, Germany
3.	*FC Portugal 2001*	Universities of Aveiro/Porto, Portugal
4.	***UvA Trilearn***	**University of Amsterdam, The Netherlands**
5.	*FC Portugal 2000*	Universities of Aveiro/Porto, Portugal
5.	*Wright Eagle*	University of Science & Technology, China
7.	*FC Tripletta*	University of Keio, Japan
7.	*YowAI*	Tokyo University of Electro-Communications, Japan

Table 10.5: Final standings (top eight teams) of *RoboCup-2001*.

these players generally became tired more quickly, it was their greater speed that enabled them to reach the deep ball before the opponent defenders and this usually led to a dangerous situation in front of the enemy goal. We were therefore surprised to see that only very few teams made use of heterogeneous players and consider their use as one of the main strategic advantages that *UvA Trilearn* had over most other teams. *RoboCup-2001* was won by the team that exhibited the best all-round performance (ball control as well as strategy). Although obviously disappointed to lose out on a top-three position, we were satisfied with the overall result because we had shown that we were fully competitive with all the other teams. Furthermore, we managed to defeat many strong teams that had already participated for several years. The final standings meant that *UvA Trilearn* was the best newcomer to RoboCup in the year 2001.

Apart from the regular simulation league competition, *UvA Trilearn* also took part in the *Scientific Evaluation* competition which is organized yearly to encourage the transfer of results from RoboCup to the scientific community at large. In this competition each participating team played a full-length game against a benchmark team (*FC Portugal 2000*) of which the result served as a base-line. After this, another game was played against this opponent in which the participating team had to deal with a handicap that was not known in advance (the handicap did not affect the benchmark team). This year the handicap consisted of the fact that the `dash_power_rate` was significantly lower when a player tried to dash in the upper half of the field as seen on the *soccer monitor*. As a result, he would be much slower in this area. It was expected that a more adaptive team would be able to cope better with this and would respond by using the opposite wing more often. Unfortunately, *UvA Trilearn* did not finish among the top three teams in this competition[5]. Another competition in which we participated was the *Coach Substitution* competition. In this competition a fixed team was chosen (*Gemini* for group A and *AT Humboldt* for group B) which played matches against itself using different coaches for both sides. A coach was only allowed to select heterogeneous players at various field positions and could make three substitutions during the game[6]. The coach associated with the winning team would be the 'winner' of the match. This competition was organized for the first time and mainly served as a testcase for subsequent years. We participated in this competition with the same coach that we used to substitute players for our *UvA Trilearn* team and reached second place out of a total number of six coaches. It must be noted however, that the format of the competition was such that the results could not be regarded as scientifically conclusive. The coach had to select the player types before the start of the match, while he had no information about the team that he was about to 'coach'. He thus had no idea about the team's strategy and did not know how effective a faster player in a certain position would be. The outcome of a game therefore depended on how well the selected player types fitted into the unknown team strategy giving the results a random character.

[5] Apart from the top three, the full result of this competition was never announced.

[6] This is different from the standard coach competition in which the coach also gives advice to the players during the game. In order to enable the coach to work with different teams, a standard coach language is used for communication.

Chapter 11

Conclusion and Future Directions

Throughout this thesis we have described the incremental development and main features of the *UvA Trilearn 2001* robotic soccer simulation team (see also [19, 21]). As such, the thesis can be regarded as a handbook for the development of a simulated robotic soccer team. In combination with the source code [48] that we have released it provides a solid framework for new teams to build upon and it can serve as a basis for future research in the field of robotic soccer simulation. In this chapter we provide a number of concluding remarks and we summarize the main features of our team (Section 11.1). Furthermore, we will also outline several promising directions for future work (Section 11.2).

11.1 Concluding Remarks

Our main objective for the soccer simulation project was twofold. Firstly, we had to set up the project and provide a solid foundation for it which would enable others to continue after our graduation. Secondly, we had to put up a good performance at the *RoboCup-2001* robotic soccer world championship in Seattle (USA). Despite the fact that these two objectives were not completely compatible (see Section 1.3), we feel that both were met to the best of our abilities. We have designed a modular agent architecture that contains all the necessary components for a simulated soccer agent and although the available time did not allow us to implement each component in an optimal way, we were able to optimize the ones which were most crucial for the success of the team. In addition, we managed to produce a relatively simple but very effective implementation for the remaining components and as a result the team had no clear weak points. This enabled us to reach fourth place at *RoboCup-2001* thereby outperforming many teams that had already participated for several years and that had performed well at previous competitions. In the remainder of this section we will outline the main features of our team and we will discuss the way in which they have enhanced the overall performance. After this, we will mention several related aspects which have enabled us to set up a solid framework and as such have contributed to the final result.

The main features of the *UvA Trilearn 2001* robotic soccer simulation team can be summarized as follows:

- **Multi-threaded three-layer architecture** (Chapter 4). The *UvA Trilearn* agents are capable of perception, reasoning and acting. Our multi-threaded agent architecture allows the agents to use a separate thread for each of these three tasks. In this way the delay caused by I/O to and from the server is minimized so that the agents can spend the majority of their time thinking about their next action. Furthermore, the architecture of each agent consists of three layers. The threads used for perception and acting form the *Interaction Layer* which takes care of the interaction with the environment and which hides the *soccer server* details as much as possible from the other layers. On top of this, the *Skills Layer* uses the functionality offered by the *Interaction Layer* to build an abstract model of the world and to implement the various skills of each agent. The top layer is the *Control Layer* which selects the best possible action from the *Skills Layer* depending on the current world state and the current strategy of the team. The *UvA Trilearn* agent architecture provides a solid foundation for the development of a simulated soccer agent. It is highly modular and contains all the components which are necessary for a good performance in every aspect of the game.

- **Flexible agent-environment synchronization scheme** (Chapter 5). This scheme enables the agents to determine the optimal moment during a cycle for sending an action command to the server. As a result, the agents are capable of performing an action in each cycle which is based on the most recent information about the state of the world when possible. This has a significant influence on the performance of the team. Compared to many other teams it was clear that our agents were capable of executing a larger number of actions since no action opportunities were missed. This often allowed us to gain an advantage over the opponents. Furthermore, our agents were always able to respond quickly to changes in the environment due to the fact that the synchronization method used maximizes the chance of basing an action choice on visual information from the current cycle.

- **Accurate methods for object localization and velocity estimation** (Chapter 6). The agent world model can be regarded as a probabilistic representation of the world state based on past perceptions. It contains information about all the objects on the soccer field as well as various methods which use the low-level world state information to derive higher-level conclusions. For each object an estimation of its position and velocity are stored (among other things) together with a confidence value indicating the reliability of the estimate. By integrating the known *soccer server* dynamics into a *particle filter* algorithm we were able to compute very accurate estimates for the positions and velocities of dynamic objects. This increased the performance of the team, since the agents could base their reasoning process on a more accurate world state representation.

- **Layered skills hierarchy** (Chapter 7). The skills which are available to the *UvA Trilearn* agents can be divided into different layers which together form a hierarchy of skills. The layers are hierarchical in the sense that the skills in each layer use skills from the layer below to generate the desired behavior. The skills in the bottom layer can be specified in terms of basic *soccer server* action commands. This framework has made it possible to reason about the various skills that an agent can perform at a high level of abstraction instead of having to deal directly with low-level server commands. In our current implementation, the *UvA Trilearn* agents are capable of performing many skills which gives them a large number of action alternatives in a given situation. Despite the fact that all skills have been hand-coded, their implementation has proved to be very effective.

- **Scoring policy** (Chapter 8). This policy enables the agents to determine the optimal target point in the goal together with an associated probability of scoring when the ball is shot to this point in a given situation. It is partly based on an approximate method that we have developed for learning the relevant statistics of the ball motion which can be regarded as a geometrically constrained continuous-time Markov process [22, 49]. Since scoring goals is one of the main objectives in a soccer game, this scoring policy has proved to be very useful.

- **Effective team strategy** (Chapter 7). To select an appropriate action, the *UvA Trilearn* agents make a distinction between *active* and *passive* situations depending on whether they currently have an active role in the game or not. If an agent is in a passive situation he moves to a strategic position on the field in anticipation of becoming active again. If an agent is in an active situation he chooses an action based on the current position of the ball and on the positions of other players.

The *UvA Trilearn* team uses a 4-3-3 formation which has proved to be very effective. The main philosophy is to keep the ball moving quickly from each player to the next and preferably in a forward direction. In addition, the agents often try to cut through the opponent's defense by passing the ball into the depth in front of the wing attackers at the side of the field. In this way, the team often succeeds in moving the ball rapidly to the enemy half of the field thereby disorganizing the opponent team. Furthermore, the effectiveness of this strategy is greatly enhanced by the fact that we use heterogeneous players on the wings. The greater speed of these players as compared to default players often enables them to reach the deep ball before the opponent defenders and this usually leads to a dangerous situation in front of the enemy goal.

The features mentioned above have clearly formed the basis of the success of *UvA Trilearn* at *RoboCup-2001*. However, the following important aspects have also contributed positively to the final result:

- **Incremental development** (Appendix A.2). Throughout the project we have consequently followed an incremental software development approach. The main advantage of this approach was that we had a working system at all times that could be tested and compared to previous versions. This helped us meet the deadlines provided by the competitions in which *UvA Trilearn* participated and enabled us to build a large system (±29,000 lines of code) in a relatively short period of time.
- **Study of related work and the *soccer server* simulation environment** (Chapters 2 and 3). During the initial stages of the project much time was spent on studying literature on the subject of multi-agent systems and on simulated robotic soccer in particular. This has enabled us to become familiar with the robotic soccer domain and has provided us with a great deal of knowledge that has been very useful throughout the project. Furthermore, we studied every feature of the *soccer server* and wrote small test programs to observe its behavior. In this way, we developed a thorough understanding of how the simulation worked and during the remainder of the project this proved to be worthwile. It has enabled us to tune our low-level implementation optimally to the characteristics of the server[1] and has led to faster debugging of low-level algorithms.
- **Software Engineering Issues** (Appendix A). To facilitate future use of our code by others (and by ourselves) much attention during the project has been focused on software engineering issues. We have consequently followed an object-oriented design methodology leading to a logical class hierarchy and highly modular code. Here it must be noted that the distribution of tasks between the project members was such that it preserved the conceptual integrity of the system (Appendix A.3). Furthermore, we extensively documented our code (±8,000 lines of documentation) and developed a multi-level log system which accelerated the debugging process considerably (Appendix A.4).

11.2 Future Work

The *soccer server* simulation environment contains many challenges which make it an interesting domain in which to conduct research. Throughout this thesis we have presented our solutions to various problems which are inherent in the simulated robotic soccer domain. However, although *UvA Trilearn* peformed well at the *RoboCup-2001* world championship, it is still possible to improve the team in many ways. In this section we will outline several promising directions for future work.

[1] This is not necessarily positive from a scientific perspective, but essential for the success of the team. It is important to realize that in order to create a standard simulation testbed, several choices must be made concerning the implementation of realistic models. The algorithms developed in the domain will then always be tuned to these choices. If the simulation changes, the general properties of many solutions will still be correct, but the details of their implementation have to be altered. This is the same for real-world algorithms: they will no longer produce correct results if the laws of physics change.

- **Multiple pieces of information in a single particle**. It is possible to further improve the position, orientation and velocity estimates for dynamic objects in the world model of the agents by including each of them into a single particle. The resulting particle set can then be updated by propagating the particles from each cycle to the next based on the current observation and in case of the agent himself on the action that he has performed. Note that one will need more particles for this filter due to the increased dimensionality of the state space.

- **Integrating uncertainty in visual perceptions into the confidence of object information**. In our current implementation, the world model of the agents does not explicitly store the uncertainty in visual perceptions to indicate the reliability of object information. This uncertainty is only implicitly included into the position and velocity estimates themselves as a result of particle filtering. The only reliability measure that is currently present comes in the form of a confidence value for each object that represents the time at which this object was last observed. If this confidence value drops below a certain threshold, then the object information is neglected in the reasoning process. However, it would be more accurate to also include the perceptual uncertainty into the reliability measure. This can be done, for example, by incorporating the variance of a perception into the confidence value and letting it propagate with time. In addition, it is possible to use the resulting confidence value more explicitly in the reasoning process.

- **The use of learning techniques for implementing skills**. In our current implementation, all the skills which are available to the agents have been hand-coded. They make use of several configurable parameters which have been given a value based on observations made during practice matches. A more principled way to implement the skills would be to *learn* them using a machine learning algorithm. One could use, for example, *reinforcement learning*. The main advantage of this approach is that it provides a way of programming the agents by means of reward and punishment without needing to specify how a task has to be achieved. However, the complexity of the robotic soccer domain (huge state space, many possible actions and strategies, partial observability of state information, etc.) make the use of traditional reinforcement learning methods difficult. Following [79], a possible way to tackle this complexity is to define sequences of basic commands instead of separate ones in order to reduce the number of actions and decisions available to the agents.

- **Stronger commitment to previous decisions**. In our current implementation, an agent commits to only a single action in each cycle ('weak binding') instead of generating a 'plan' consisting of a sequence of actions over a number of cycles. This means that if an agent executed part of a ball-interception skill in the previous cycle there is no guarantee that he will continue to execute the next part in the current cycle. The situation is completely reevaluated before each action opportunity and an action is selected based on the current situation. This has the advantage that in dynamic domains, such as that provided by the *soccer server*, the agents are flexible to changes in the environment since they always select the best possible action based on the current state of the world. However, a disadvantage is that it can lead to oscillations between different skills in consecutive cycles[2]. A better solution would be for the agent to commit to a certain skill for a longer period of time. It is important that this time period is not too long however, since the dynamic nature of the *soccer server* simulation environment requires the agent to remain flexible to environmental changes.

- **An adaptive scoring policy**. Our current solution to the scoring problem is based on a fixed goalkeeper and a fixed distance to the goal. This solution can be improved by incorporating an extra feature into the model representing the distance to the scoring point as was outlined in Section 8.6. Furthermore, the probability of passing the goalkeeper should be *adaptive* for different goalkeepers. This means that the model should incorporate information about the current opponent goalkeeper instead of using that of a particular team. The desired case would be to let the model adapt itself

[2]Note that in practice this will not often happen due to the fact that the environment changes gradually.

during the game, using little prior information about the current goalie. This is a difficult problem because learning must be based on only a few scoring attempts. It is therefore important to extract the most relevant features and to parametrize the intercepting behavior of the opponent goalkeeper in a compact manner that permits on-line learning. A possible way to do this might be through the use of statistics collected by the coach.

- **An extended positioning mechanism**. In our current implementation, the strategic position of an agent depends only on the agent's home position inside the current formation and on the position of the ball which serves as an attraction factor. It is possible to extend this positioning mechanism by incorporating additional factors such as the positions of other players. Following [115], the strategic position of an agent can then be computed according to a multiple-objective function which maximizes the distance from opponents and passive teammates (repulsion factors) and minimizes the distance to the enemy goal and to the teammate with the ball (attraction factors).

- **Switching roles inside dynamically changing formations**. In our current implementation, the *UvA Trilearn* team always uses the same 4-3-3 formation in which each agent has a fixed role. It is possible to extend this implementation by defining multiple formations (4-4-2, 3-5-2, etc.) and by switching among these formations depending on the score in the game or the tactic of the opponent team. Furthermore, it is desirable that the agents are capable of switching roles inside a formation when the situation asks for it. This can happen, for example, if an agent moves very far from his strategic position when intercepting the ball. Two agents could then decide to switch roles if the sum of the distances to their respective strategic positions would decrease when doing so. In general, the decision *when* to switch roles should depend on the resources which are available for filling a particular role. Assume, for example, that agent a_1 has role r_1 and agent a_2 has role r_2. In this case, a clear reason for switching roles would be if a_1 has more of the resources necessary for filling r_2 than a_2 does and likewise a_2 for r_1. When switching roles it is important that the agents involved somehow agree upon making the switch in order to avoid that a single role will be filled by multiple agents. Another problem arises when the switch involves heterogeneous players with different abilities. For certain roles, these abilities might not have the desired effect.

- **Exploiting the potential of heterogeneous players at various field positions**. In our current implementation, only five roles inside the formation of the team are filled by non-default players: two wing attackers, a central attacker and two wing defenders. The player types for these roles are selected according to a utility function that combines the values of various player parameters. It must be noted that the choice of roles to be filled by heterogeneous players is based on observations made during practice games and on the assumption that the team uses a 4-3-3 formation which is the standard for attacking along the wings. A more general solution would be to develop a method for predicting the utility of a player type depending on the role of this player. In this way, it can be determined which player types are best suited for which roles in a given formation. A possible approach would be to use machine learning techniques (e.g. a genetic algorithm) to learn an effective distribution of heterogeneous players on the field.

- **Multi-agent modeling using a coach agent**. In our current implementation, individual agent decisions are often affected by the behavior of opponent players, but strategic decisions for the team as a whole are mostly fixed. Ideally however, the team strategy should be adjusted in response to adversarial behavior. A promising approach is to use the coach agent for this purpose. The coach receives noise-free global information about the state of the world and has less real-time demands than the players. As a result, he can spend more time deliberating over strategies. The online coach is therefore a good tool for analyzing the strengths and weaknesses of the opponent team and for giving advice on the best possible strategy. To this end, the coach must be able to classify enemy behavior based on certain features. He can then decide which strategy is most appropriate for the given behavior class. The coach can determine, for example, the playing style of the opponent team and based on this he can recommend to change the formation or to switch player types for a certain role inside the current formation. In turn, the players must know how to interpret the coach's advice and how to act upon it.

Appendix A

Software Engineering Aspects

Creating a complete multi-agent system, such as a simulated robotic soccer team, is not a straightforward task. The main difficulty arises from the fact that such a system consists of many different components which have to operate together in an appropriate way. Furthermore, building each separate component is a difficult task in itself. It is obvious that a project of this scale cannot become a success if it is not well organized. Throughout the project we have therefore paid much attention to the software engineering aspects of our implementation. In order to facilitate future use of our code by others we have set up a software architecture that allows for the various components to be combined in a clear and orderly manner. This has enabled us (and will enable others) to extend and debug large amounts of code in a structured way and made it possible to follow an incremental software development approach. Unfortunately, current literature in the field of simulated robotic soccer pays little attention to the subject of software engineering ([53] is an exception). We therefore feel that it is appropriate to provide some details in this appendix. We will mainly focus on problems that typically arise in large software projects and on how we have tried to avoid them. This appendix is organized as follows. In Section A.1 we discuss several implementation issues such as our documentation system and the programming language that we used. Section A.2 is devoted to *incremental development*, the software development approach that we have followed throughout the project. Section A.3 shortly addresses the issue of manpower distribution and presents a model for dividing tasks between various programmers which protects the conceptual integrity of the system. Finally, Section A.4 contains a description of a tool that we have developed to speed up the debugging process.

A.1 Implementation Issues

Two important choices for the implementation of our team were which programming language to use and for which platform it should be developed. We have chosen to use the *C++* programming language [105] and to compile the code using a *gcc* compiler. *C++* was mainly chosen for reasons of performance and because the language supports an object-oriented approach. Especially the performance aspect is crucial in the *soccer server* simulation environment due to the real-time nature of the domain. We have consequently followed an object-oriented design methodology leading to a logical class hierarchy and highly modular code. The team has been built and tested for the Linux (Red Hat) operating system[1] since it represents the standard which is used during competitions. Furthermore, most soccer simulation teams have been developed for this platform and have released their Linux binaries on the web. Using Linux thus makes

[1] It also works under Solaris after making some changes to compensate for the differences in the *gcc* libraries of both systems.

it easier to test against these teams. The multi-threaded agent architecture described in Section 4.2 has been implemented using POSIX threads [12] as they appeared intuitively appropriate for the requirements and (assuming implementation in *C++*) have native support in the Linux operating system.

To facilitate future use of our code by others much time during the implementation has been spent on extensively documenting our code. For this we used the documentation system *Doxygen* [114]. This system can be used for different programming languages (*C++*, *Java*, *IDL* and *C*) to automatically generate a reference manual from a set of documented source files. It is currently also used as the documentation system for the GNU Standard *C++* Library and the Red Hat Linux Packaging Manager. To use the system one needs to document various parts of the program (classes, class variables, methods, etc.) by describing their functionality at the location where they are defined. This has the advantage that the documentation is located close to the definition which makes it easier to keep it consistent with the source code. *Doxygen* then extracts this information and generates an on-line documentation browser (in HTML) and/or an off-line reference manual. There is also support for generating output in RTF (MS-Word), PostScript, hyperlinked PDF, compressed HTML and Unix man pages. *Doxygen* can also be used to extract the code structure from a set of source files by visualizing the relations between the various elements through automatically generated dependency diagrams, inheritance diagrams and collaboration diagrams. These functionalities were all utilized during the development of the *UvA Trilearn* soccer simulation team.

If several people (in our case two) are working on the same project, version management becomes an important issue. The version control system that we have used during the development of our team is *CVS* (Concurrent Versions System). This widely used system allows one to store old versions of files and to keep a log of who made changes and when and why they were made. Unlike simpler systems (*RCS*, *SCCS*), *CVS* does not only operate on one file or one directory at a time, but on hierarchical collections of directories consisting of several version controlled files. *CVS* has helped us to manage different versions of our team and to control the concurrent editing of source files among multiple authors. It has enabled us to store various releases in a convenient way and made it possible to return to a previous version if some extension or update did not show the intended result. Currently, the implementation of our team consists of approximately 29,000 lines of code of which about 8,000 are documentation lines[2]. This large amount of code, together with the fact that many extensions are required each year to improve the team and to accommodate changes made to the *soccer server* simulation environment, indicates that it is necessary to keep the code well structured. Following an incremental approach and using the above-mentioned tools makes it easier to achieve this.

A.2 Incremental Development

The main problem when building a large system, such as a simulated robotic soccer team, is that its conceptual structure is too complex to be completely and accurately specified in advance and too complex to be built without faults. We have therefore implemented our team according to a software development technique called *incremental development* [8, 62, 68]. This approach dictates that a system should first be made to run, even though it does nothing useful except for creating the proper set of dummy objects. Each object is then gradually refined by adding more and more functionality until the system is fully 'grown'. The main advantage of this approach is that it gives one a working system at all times (which is good for moral) that can be tested and compared to earlier versions. Furthermore, it necessitates a top-down design since it amounts to top-down growing of the software.

We have applied the incremental approach by first creating an extremely simple system that consisted of three threads that could perform the basic loop of receiving information from the server, processing this

[2] This could be checked using *Doxygen* that also outputs the formatted source code without the comments.

information and sending an action to the server. Each component in this system was built in a simple way only performing its task at a very elementary level. Some components would even do nothing just being implemented as void subroutines taking their correct place in the overall architecture. Although this initial system clearly did not do much, it certainly did it correctly and it could be regarded as our first 'working' version. We then progressively refined this simple implementation by extending the functionality of the different components one by one while keeping the architecture as a whole intact. We started by refining the *ActHandler* component (see Figure 4.2) and by implementing our *Flexible External Windowing* synchronization scheme (see Section 5.3.4). This made it possible to send a command to the server in each cycle and enabled the agent to perform his first basic actions. We then implemented the *SenseHandler* component which parsed the messages that arrived from the server and stored the information in a simple world model. After this, the world model was gradually extended and using the information contained in it we started to create a version of our team that could actually play a game of soccer. This led to the *De Meer 5* soccer simulation team (see Section 9.5.1) that was mainly used for testing our low-level implementation. Subsequently, the action selection policy for the agents has been progressively refined to create more sophisticated versions of our team. Following this approach had several advantages. Firstly, we always had a working system and as a result were always sure to have a team ready for the competitions in which we participated. *Incremental development* can thus be seen to protect against schedule overruns (at the cost of possible functional shortfall). Furthermore, the approach enabled us to thoroughly test the various components of the system and made it easier to locate faults since we always knew that they had to originate from the last refinement step.

A.3 Manpower Distribution

An important aspect of a software project is how to divide the work over the different (in our case two) team members. A conventional method is to partition the task equally over the available people which are each responsible for the design and implementation of their part. This has several disadvantages however. Firstly, it usually leads to systems which lack *conceptual integrity* (i.e. unity of design). It is better if a system reflects one set of design ideas, than to have one that contains many good but independent and uncoordinated ideas. Large programming systems are usually not conceptually integrated since the design is separated into many tasks done by many men. To achieve conceptual integrity, a design must proceed from one mind or a small group of agreeing minds. Every part of the system must reflect the same philosophies. Furthermore, partitioning a task among multiple people occasions extra communication effort in the form of training and intercommunication. Especially the extra intercommunication, which is usually large for software projects and which increases quadratically ($n(n-1)/2$) with the number of team members, quickly dominates the decrease in individual task time brought about by the partitioning[3]. A major part of the cost of a system lies in this communication and in correcting the ill effects of miscommunication. The number of minds that have to be coordinated thus affects the total cost and quality of the system. This suggests that you want a system to be built by as few minds as possible [8]. However, schedule deadlines usually dictate that building a large system requires many hands. This leads to a difficult problem. For efficiency and conceptual integrity of a system one needs a small team of good people to design and implement it. Yet, for large systems one desires a way to bring considerable manpower to bear so that the system can be finished on time. How can these two needs be reconciled?

A proposal by Harlan Mills [61] offers a creative solution to this problem that we have consequently followed throughout the project. He suggests that a large system should be built by a small team (or several small teams) of software developers which is organized like a surgical team. The 'surgeon' (Mills calls him the chief-programmer) is the leader of the team and is responsible for the end product, whereas

[3] [8] actually shows that adding manpower to a late software project makes it later (Brooks' law).

his assistants are there to help him design and construct the system. An example is the 'co-pilot' who can be seen as the alter ego of the surgeon: he knows the complete design and all the code that has been written and gives advice on possible improvements. The surgeon can use this information for making changes to the code, but is not bounded to do so. Other examples are the 'editor' who reviews the code (including documentation) and the 'tester' who is responsible for testing the system. In this setup, the conceptual integrity of the system lies in the hands of the surgeon. This means that instead of each team member 'cutting' away at the problem, only one person does the real cutting and the others give him support that will enhance his effectiveness and productivity. This chief-programmer surgical team organization offers a way to get the product integrity of one or a few minds and the total productivity of many helpers with radically reduced communication. We have adopted this approach during the implementation of the *UvA Trilearn* team: one team member played the role of the surgeon, whereas the other performed the combined tasks of the other members of the surgical team[4]. In combination with the incremental approach described in Section A.2 this has enabled us to build a large system in a relatively short period of time.

A.4 Multi-Level Log System

During the development of autonomous agents it is important for the designer to be able to trace their action choices and to understand why they act as they do. When an agent does something unexpected or undesirable, the designer needs a way to isolate precisely why the agent took such an action. In the *soccer server* simulation environment this is not straightforward however. This is caused by the fact that an enormous amount of data passes through the system when a game is played which makes it difficult to find the source of a problem. Each agent is controlled by a separate client process and on average receives and parses 22 messages (10 **sense_body** messages, 7 **see** messages and 5 **say** messages) from the server in one second. As a result, it is impossible to keep track of the complete data flow that is running through the system during a 10-minute match. Printing information for debugging purposes, for example, quickly leads to huge data files and drawing relevant conclusions from these data is a very time-consuming effort. This makes it difficult to identify what exactly caused an agent to act as it did in a given situation. When an agent performs an unexpected action this can be caused, for example, by an error in the processing of sensory information or by a fault in the reasoning process.

To avoid being overwhelmed with data, it is clearly desirable for the observer to be able to trace only certain parts of the agent program at a level of abstraction that he chooses. To this end, we have implemented a multi-level log system based on the ideas presented in [81]. This system enables the user to specify the abstraction level from which he desires information and to log this information for debugging purposes. The programmer can then trace the recorded information to find why the agent took each of his actions and to identify which parts of the agent have to be altered. The key to this approach is the fact that the relevant agent information is organized in layers which are defined by the user. These layers can correspond to agent-specific information (e.g. his internal state) or to higher-level reasoning processes for a particular purpose (e.g. skills execution or action selection). In general, there is far too much information available to display all of it at all times. Furthermore, it is usually not necessary to receive the information from all the layers during the debugging process. The imposed hierarchy therefore allows the user to select at which level of detail he or she would like to probe into the agent. This makes it possible to zoom into the details that are important for the current situation and to hide the details from the other layers.

We have implemented this multi-level log system in the form of a *Logger* class (see Section 4.3). This class allows the programmer to specify the abstraction level(s) from which he desires information and contains

[4] A similar recipe was followed for the effort of writing this thesis. The only difference was that the authors switched roles for this task.

an output stream for writing this information (usually a file). During his execution, the agent logs certain information by calling a method from the *Logger* class. The information is written to the specified output stream only if the number associated with the layer from which the information originates falls within the range of layers from which the user desires information. Table A.1 shows the information hierarchy that we have defined in our implementation of the log system. Here it must be noted that we have not completely specified each level for reasons of clarity. Instead, we only show ranges of levels which correspond to the main components of the agent. When a match is replayed using the *logplayer*, the gathered information for a certain level can be used to determine why an agent executed a specific action in a given situation. Throughout the project, our log system has been a crucial tool for the development of the *UvA Trilearn* agents which has accelerated the debugging process considerably.

Levels	**Information**
1-20	communication with the server
21-30	updates to the world model
31-60	the state of the world model
61-100	skills execution
100-150	action selection

Table A.1: Information hierarchy for our multi-level log system.

Bibliography

[1] T. Andou. *Refinement of Soccer Agents' Positions Using Reinforcement Learning.* In H. Kitano, editor, *RoboCup-97: Robot Soccer World Cup I*, pages 373–388. Springer Verlag, Nov. 1998.

[2] J. Bacon. *Concurrent Systems.* International Computer Science Series. Addison Wesley, 2nd edition, 1998.

[3] M. Badjonski, K. Schroter, J. Wendler, and H. D. Burkhard. *Learning of Kick in Artificial Soccer.* In W. van der Hoek, editor, *Proceedings of the European RoboCup Workshop*, May 2000.

[4] R. Barman, S. Kingdon, J. Little, A. Mackworth, D. Pai, M. Sahota, H. Wilkinson, and Y. Zhang. *Dynamo: Real-Time Experiments with Multiple Mobile Robots.* In *Proceedings of the IEEE Intelligent Vehicles Symposium*, pages 261–266, Tokyo, July 1993.

[5] A. G. Barto, S. J. Bradtke, and S. P. Singh. *Learning to Act using Real-Time Dynamic Programming.* *Artificial Intelligence*, 72:81–138, 1995.

[6] A. H. Bond and L. Gasser. *An Analysis of Problems and Research in DAI*, pages 3–35. Morgan Kaufmann Publishers Inc., Los Angeles, CA, 1988.

[7] M. Bowling, P. Stone, and M. Veloso. *Predictive Memory for an Inaccessible Environment.* In *Proceedings of the IROS-96 Workshop on RoboCup*, pages 28–34, Osaka, Nov. 1996.

[8] F. P. Brooks, Jr. *The Mythical Man-Month: Essays on Software Engineering.* Addison Wesley, Berkeley, CA, 1995.

[9] R. A. Brooks. *Intelligence Without Reason.* In *Proceedings of the 12th International Joint Conference on Artificial Intelligence (IJCAI-91)*, pages 569–595, Sydney, 1991. Morgan Kaufmann.

[10] H. D. Burkhard, M. Hannebauer, and J. Wendler. *AT Humboldt - Development, Practice and Theory.* In H. Kitano, editor, *RoboCup-97: Robot Soccer World Cup I*, pages 357–372. Springer Verlag, 1998.

[11] H. D. Burkhard, T. Meinert, H. Myritz, and G. Sander. *AT Humboldt Team Description.* In M. Veloso, E. Pagello, and H. Kitano, editors, *RoboCup-99: Robot Soccer World Cup III.* Springer Verlag, 2000.

[12] R. Butenhof. *Programming with POSIX Threads.* Addison-Wesley, 1997.

[13] M. Butler, M. Prokopenko, and T. Howard. *Flexible Synchronisation within the RoboCup Environment: a Comparative Analysis.* In P. Stone, T. Balch, and G. Kraetszchmar, editors, *RoboCup-2000: Robot Soccer World Cup IV*, pages 119–128. Springer Verlag, Berlin, 2001.

[14] M. G. Catlin. *The Art of Soccer.* Soccer Books, 1990.

[15] M. Collins and E. E. Aldrin, Jr. *Apollo Expeditions to the Moon.* NASA SP, 1975.

[16] E. Corten. *The WindMill Wanderers Source Code.* University of Amsterdam, 1998. Not publicly available.

[17] E. Corten and F. Groen. *Team description of the UvA Team.* In *RoboCup-99 Team Descriptions for the Simulation League*, pages 149–153. Linköping University Press, 1999.

[18] E. Corten and E. Rondema. *Team description of the Windmill Wanderers.* In *Proceedings of the Second RoboCup Workshop*, pages 347–352, Paris, 1998.

[19] R. de Boer, J. Kok, and F. Groen. *The UvA Trilearn 2001 Robotic Soccer Simulation Team. BNVKI Newsletter*, 18(4), Aug. 2001.

[20] R. de Boer, J. Kok, and F. Groen. *UvA Trilearn 2001 Research Abstract.* Part of qualification material for RoboCup-2001. At http://www.science.uva.nl/~jellekok/robocup, Apr. 2001.

[21] R. de Boer, J. Kok, and F. Groen. *UvA Trilearn 2001 Team Description.* In *Robocup-2001: Robot Soccer World Cup V.* Springer Verlag, Berlin, 2002.

[22] R. de Boer, J. Kok, N. Vlassis, and F. Groen. *Towards an Optimal Scoring Policy for Simulated Soccer Agents*, Jan. 2002. Submitted to the RoboCup-2002 workshop in Fukuoka (Japan). At http://www.science.uva.nl/~jellekok/robocup.

[23] T. Dean and M. Boddy. *An Analysis of Time-Dependent Planning.* In *Proceedings of the Seventh National Conference on Artificial Intelligence (AAAI-88).* Morgan Kaufmann, 1988.

[24] K. Dorer. *Behavior Networks for Continuous Domains using Situation-Dependent Motivations.* In *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI'99)*, pages 1233–1238, Stockholm, 1999. Morgan Kaufmann.

[25] K. Dorer. *Extended Behavior Networks for the Magma Freiburg Team.* In *RoboCup-99 Team Descriptions for the Simulation League*, pages 79–83. Linköping University Press, 1999.

[26] K. Dorer. *The Magma Freiburg Soccer Team.* In M. Veloso, E. Pagello, and H. Kitano, editors, *RoboCup-99: Robot Soccer World Cup III.* Springer Verlag, 2000.

[27] K. Dorer. *Personal correspondence during RoboCup-2001.* Seattle, 2001.

[28] A. Doucet, N. de Freitas, and N. Gordon. *Sequential Monte Carlo Methods in Practice.* Springer Verlag, 2001.

[29] A. E. Elo. *The Rating of Chessplayers, Past & Present.* Arco, 1978.

[30] W. Feller. *An Introduction to Probability Theory and Its Applications*, volume 1. John Wiley & Sons, Inc., New York, 1957.

[31] W. Feller. *An Introduction to Probability Theory and Its Applications*, volume 2. John Wiley & Sons, Inc., New York, 1967.

[32] E. Foroughi, F. Heintz, S. Kapetanakis, K. Kostiadis, J. Kummeneje, I. Noda, O. Obst, P. Riley, and T. Steffens. *RoboCup Soccer Server User Manual: for Soccer Server version 7.06 and later*, 2001. At http://sourceforge.net/projects/sserver.

[33] D. Garlan and M. Shaw. *An Introduction to Software Architecture. Advances in Software Engineering and Knowledge Engineering*, volume 1, 1995.

[34] F. Girault and S. Stinckwich. *Footux Team Description: A Hybrid Recursive Based Agent Architecture.* In *RoboCup-99 Team Descriptions for the Simulation League*, pages 31–35. Linköping University Press, 1999.

[35] P. Goetz. *Attractors in Recurrent Behavior Networks.* PhD thesis, State University of New York at Buffalo, 1997.

[36] H. Hu, K. Kostiadis, M. Hunter, and M. Seabrook. *Essex Wizards '99 Team Description.* In M. Veloso, E. Pagello, and H. Kitano, editors, *RoboCup-99: Robot Soccer World Cup III.* Springer Verlag, 2000.

[37] H. Hu, K. Kostiadis, and Z. Liu. *Coordination and Learning in a Team of Mobile Robots.* In *Proceedings of the IASTED Robotics and Applications Conference*, California, Oct. 1999.

[38] C. Hughes. *Soccer Tactics and Teamwork.* EP Publishing Ltd., 1973.

[39] M. Hunter, K. Kostiadis, and H. Hu. *A Behavior-Based Approach to Position Selection for Simulated Soccer Agents.* In W. van der Hoek, editor, *Proceedings of the European RoboCup Workshop*, May 2000.

[40] P. L. Jakab. *Visions of a Flying Machine: The Wright Brothers and the Process of Invention.* Smithsonian Institution Press, Washington D.C., 1990.

[41] L. P. Kaelbling, A. R. Cassandra, and M. L. Littman. *Acting Optimally in Partially Observable Stochastic Domains.* In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*, 1994.

[42] L. P. Kaelbling and S. J. Rosenschein. *Action and Planning in Embedded Agents.* In P. Maes, editor, *Designing Autonomous Agents: Theory and Practice from Biology to Engineering and Back*, pages 35–48. MIT/Elsevier, 1990.

[43] R. Kalman. *A New Approach to Linear Filtering and Prediction Problems. Journal of Basic Engineering*, pages 35–46, 1966.

[44] H. Kitano and M. Asada. *RoboCup Humanoid Challenge: That's One Small Step for A Robot, One Giant Leap for Mankind.* In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS-98)*, 1998.

[45] H. Kitano, M. Asada, Y. Kuniyoshi, I. Noda, and E. Osawa. *RoboCup: The Robot World Cup Initiative.* In *Proceedings of the IJCAI-95 Workshop on Entertainment and AI/Alife*, 1995.

[46] H. Kitano, M. Asada, Y. Kuniyoshi, I. Noda, and E. Osawa. *RoboCup: The Robot World Cup Initiative.* In *Proceedings of the First International Conference on Autonomous Agents (Agent-97)*, 1997.

[47] J. Kok and R. de Boer. *The Official UvA Trilearn Website.* University of Amsterdam, 2001. At http://www.science.uva.nl/~jellekok/robocup.

[48] J. Kok and R. de Boer. *The UvA Trilearn 2001 Source Code.* University of Amsterdam, 2001. At http://www.science.uva.nl/~jellekok/robocup.

[49] J. Kok, R. de Boer, and N. Vlassis. *Towards an Optimal Scoring Policy for Simulated Soccer Agents.* In *Proceedings of the Seventh International Conference on Intelligent Autonomous Systems (IAS-7)*, Marina del Rey, California, Mar. 2002. To appear as a short paper.

[50] K. Kostiadis and H. Hu. *Essex Wizards*. In *RoboCup-99 Team Descriptions for the Simulation League*, pages 17–26. Linköping University Press, 1999.

[51] K. Kostiadis and H. Hu. *Reinforcement Learning and Cooperation in a Simulated Multi-Agent System*. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots & Systems (IROS '99)*, Korea, Oct. 1999.

[52] K. Kostiadis and H. Hu. *A Multi-Threaded Approach to Simulated Soccer Agents for the RoboCup Competition*. In M. Veloso, E. Pagello, and H. Kitano, editors, *RoboCup-99: Robot Soccer World Cup III*. Springer Verlag, 2000.

[53] K. Kostiadis, M. Hunter, and H. Hu. *The Use of Design Patterns for the Development of Multi-Agent Systems*. In *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics (SMC2000)*, Oct. 2000.

[54] M. L. LaBlanc and R. Henshaw. *The World Encyclopedia of Soccer*. Visible Ink Press, 1994.

[55] J.-C. Latombe. *Robot Motion Planning*. Kluwer Academic Publishers, Boston, 1991.

[56] J. N. Lau and L. P. Reis. *FC Portugal Most Interesting Research Overview*. An overview of the FC Portugal team. At http://www.ieeta/pt/robocup, 2000.

[57] M. Lenz, B. Bartsch-Sporl, H. Burkhard, and S. Wess. *Case Based Reasoning Technology*. In *From Foundations to Applications*, LNAI 1400. Springer Verlag, 1998.

[58] J. Lindeberg. *Eine Neue Herleitung des Exponentialgesetzes in der Wahrscheinlichkeitsrechnung*. *Mathematische Zeitschrift*, 15:211–225, 1922.

[59] A. Mackworth. *On Seeing Robots*. In *Computer Vision: Systems, Theory, and Applications*, pages 1–13. World Scientific Press, Singapore, 1993.

[60] P. Maes. *The Dynamics of Action Selection*. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence (IJCAI-89)*, Detroit, 1989. Morgan Kaufmann.

[61] H. Mills. *Chief Programmer Teams, Principles and Procedures*. Technical Report FCS 71-5108, IBM Federal Systems Division, Gaithersburg, MD, 1971.

[62] H. Mills. *Top-Down Programming in Large Systems*. In R. Rustin, editor, *Debugging Techniques in Large Systems*. Prentice Hall, Englewood Cliffs, NJ, 1971.

[63] J. Murray, O. Obst, and F. Stolzenburg. *RoboLog Koblenz 2000*. In W. van der Hoek, editor, *Proceedings of the European RoboCup Workshop*, May 2000.

[64] J. Murray, O. Obst, and F. Stolzenburg. *Towards a Logical Approach for Soccer Agents Engineering*. In P. Stone, T. Balch, and G. Kraetszchmar, editors, *RoboCup-2000: Robot Soccer World Cup IV*. Springer Verlag, Berlin, 2001.

[65] M. Ohta. *Gemini - Learning Cooperative Behaviors Without Communicating*. In *RoboCup-99 Team Descriptions for the Simulation League*, pages 36–39. Linköping University Press, 1999.

[66] A. Papoulis. *Probability, Random Variables and Stochastic Processes*. McGraw-Hill, 3rd edition, 1991.

[67] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design: the Hardware/Software Interface*. European adaptation. Morgan Kaufmann Publishers, San Francisco, CA, 1994.

[68] R. S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, 4th edition, 1997.

[69] M. Prokopenko. *Situated Reasoning in Multi-Agent Systems.* In *The AAAI-99 Spring Symposium on Hybrid Systems and AI*, pages 158–163, Stanford, 1999.

[70] M. Prokopenko and M. Butler. *Tactical Reasoning in Synthetic Multi-Agent Systems: a Case Study.* In *Proceedings of the IJCAI-99 Workshop on Nonmonotonic Reasoning, Action and Change*, pages 57–64, Stockholm, 1999.

[71] M. Prokopenko, M. Butler, and T. Howard. *Cyberoos 2000: Experiments with Emergent Tactical Behavior.* University of Melbourne, 2000. At http://www.cmis.csiro.au/aai/RoboCup/Publications.

[72] M. Prokopenko, M. Butler, and T. Howard. *On Emergence of Scalable Tactical and Strategic Behavior.* In P. Stone, T. Balch, and G. Kraetszchmar, editors, *RoboCup-2000: Robot Soccer World Cup IV*. Springer Verlag, Berlin, 2001.

[73] M. Prokopenko, M. Butler, W. Y. Wong, and T. Howard. *Cyberoos '99: Tactical Agents in the RoboCup Simulation League.* In M. Veloso, E. Pagello, and H. Kitano, editors, *RoboCup-99: Robot Soccer World Cup III*. Springer Verlag, 2000.

[74] M. Prokopenko, R. Kowalczyk, M. Lee, and W. Y. Wong. *Designing and Modeling Situated Agents Systematically: Cyberoos '98.* In *Proceedings of the PRICAI-98 Workshop on RoboCup*, pages 75–89, Singapore, 1998.

[75] J. R. Quinlan. *C4.5: Programs for Machine Learning.* Morgan Kaufmann, San Mateo, CA, 1993.

[76] L. P. Reis and J. N. Lau. *FC Portugal Team Description: RoboCup-2000 Simulation League Champion.* In P. Stone, T. Balch, and G. Kraetszchmar, editors, *RoboCup-2000: Robot Soccer World Cup IV*, pages 29–40. Springer Verlag, Berlin, 2001.

[77] L. P. Reis, J. N. Lau, and L. S. Lopes. *F.C. Portugal Team Description Paper.* Part of qualification material RoboCup-2000.

[78] M. Riedmiller. *Concepts and Facilities of a Neural Reinforcement Learning Control Architecture for Technical Process Control.* Journal of Neural Computing and Application, 8:323–338, 2000.

[79] M. Riedmiller, A. Merke, D. Meier, A. Hoffman, A. Sinner, O. Thate, and R. Ehrmann. *Karlsruhe Brainstormers - A Reinforcement Learning Approach to Robotic Soccer.* In P. Stone, T. Balch, and G. Kraetszchmar, editors, *RoboCup-2000: Robot Soccer World Cup IV*, pages 367–372. Springer Verlag, Berlin, 2001.

[80] M. Riedmiller, A. Merke, D. Meier, A. Hoffmann, A. Sinner, O. Thate, and C. Kill. *Karlsruhe Brainstormers 2000 - Design Principles.* Part of qualification material for RoboCup-2000.

[81] P. Riley, P. Stone, and M. Veloso. *Layered Disclosure: Revealing Agents' Internals.* In *Proceedings of the Seventh International Workshop on Agent Theories, Architectures and Languages (ATAL-2000)*, 2000.

[82] B. Ripley. *Pattern Recognition and Neural Networks.* Cambridge University Press, Cambridge, U.K., 1996.

[83] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual.* Addison Wesley, 1999.

[84] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, Englewood Cliffs, NJ, 1995.

[85] M. K. Sahota, A. K. Mackworth, R. A. Barman, and S. J. Kingdon. *Real-Time Control of Soccer-Playing Robots Using Off-Board Vision: The Dynamite Testbed.* In *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics*, pages 3663–3690, 1995.

[86] J. Schaeffer and A. Plaat. *Kasparov versus Deep Blue: The Re-Match.* *Journal of the International Computer Chess Association*, 20(2):95–101, 1997.

[87] J. Sear. *RoboBase: A Logplayer and Analysis Tool for RoboCup Logfiles.* University of Manchester, 2001. At http://www.cs.man.ac.uk/~searj6.

[88] M. Shaw and D. Garlan. *Formulations and Formalisms in Software Architecture.* *Lecture Notes in Computer Science*, volume 1000, 1995.

[89] F. Stolzenburg, O. Obst, J. Murray, and B. Bremer. *Spatial Agents Implemented in a Logical Expressible Language.* In M. M. Veloso, editor, *Proceedings of the Third International Workshop on RoboCup*, pages 205–210, Stockholm, 1999. IJCAI press.

[90] P. Stone. *Layered Learning in Multi-Agent Systems.* PhD thesis, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, Dec. 1998.

[91] P. Stone. *The CMUnited-99 Source Code.* Carnegie Mellon University, Pittsburgh, PA, 1999. At http://www-2.cs.cmu.edu/~pstone/RoboCup/CMUnited99-source.tar.gz.

[92] P. Stone, P. Riley, and M. Veloso. *The CMUnited-99 Champion Simulator Team.* In M. Veloso, E. Pagello, and H. Kitano, editors, *RoboCup-99: Robot Soccer World Cup III.* Springer Verlag, 2000.

[93] P. Stone and M. Veloso. *A Layered Approach to Learning Client Behaviors in the RoboCup Soccer Server.* *Applied Artificial Intelligence*, 12:165–188, 1998.

[94] P. Stone and M. Veloso. *Communication in Domains with Unreliable, Single-Channel, Low-Bandwidth Communication.* In *Collective Robotics*, pages 85–97, Berlin, July 1998. Springer-Verlag.

[95] P. Stone and M. Veloso. *The CMUnited-97 Simulator Team.* In H. Kitano, editor, *RoboCup-97: Robot Soccer World Cup I*, pages 387–397. Springer Verlag, Nov. 1998.

[96] P. Stone and M. Veloso. *Towards Collaborative and Adversarial Learning: A Case Study in Robotic Soccer.* *International Journal of Human Computer Studies*, 48(1):83–104, Jan. 1998.

[97] P. Stone and M. Veloso. *Using Decision Tree Confidence Factors for Multi-Agent Control.* In H. Kitano, editor, *RoboCup-97: Robot Soccer World Cup I*, pages 99–111. Springer Verlag, Nov. 1998.

[98] P. Stone and M. Veloso. *Task Decomposition and Dynamic Role Assignment for Real-Time Strategic Teamwork.* In *Intelligent Agents V - Proceedings of the Fifth International Workshop on Agent Theories, Architectures and Languages (ATAL-98)*, volume 1555, pages 293–308, Heidelberg, 1999. Springer Verlag.

[99] P. Stone and M. Veloso. *Task Decomposition, Dynamic Role Assignment and Low-Bandwidth Communication for Real-Time Strategic Teamwork.* *Artificial Intelligence*, 110(2):241–273, June 1999.

[100] P. Stone and M. Veloso. *Layered Learning and Flexible Teamwork in RoboCup Simulation Agents.* In M. Veloso, E. Pagello, and H. Kitano, editors, *RoboCup-99: Robot Soccer World Cup III.* Springer Verlag, 2000.

[101] P. Stone and M. Veloso. *Multi-Agent Systems: A Survey from a Machine Learning Perspective.* *Autonomous Robotics*, 8(3), July 2000.

[102] P. Stone, M. Veloso, and S. Achim. *Collaboration and Learning in Robotic Soccer.* In *Proceedings of the Micro-Robot World Cup Soccer Tournament*, Nov. 1996.

[103] P. Stone, M. Veloso, and P. Riley. *Team-Partitioned, Opaque-Transition Reinforcement Learning.* In *RoboCup-98: Robot Soccer World Cup II.* Springer Verlag, Berlin, 1999.

[104] P. Stone, M. Veloso, and P. Riley. *The CMUnited-98 Champion Simulator Team.* In *RoboCup-98: Robot Soccer World Cup II.* Springer Verlag, Berlin, 1999.

[105] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, 3rd edition, 1997.

[106] T. Suzuki. *Team YowAI Description.* In *RoboCup-99 Team Descriptions for the Simulation League*, pages 31–35. Linköping University Press, 1999.

[107] T. Suzuki, S. Asahara, H. Kurita, and I. Takeuchi. *Team YowAI-2000 Description.* In P. Stone, T. Balch, and G. Kraetszchmar, editors, *RoboCup-2000: Robot Soccer World Cup IV*, pages 509–512. Springer Verlag, Berlin, 2001.

[108] The RoboCup Federation. *The Official RoboCup Website.* At http://www.robocup.org.

[109] G. Thomas and R. Finney. *Calculus and Analytic Geometry.* Addison Wesley, 9th edition, 1999.

[110] S. Thrun, D. Fox, W. Burgard, and F. Dellaert. *Robust Monte Carlo Localization for Mobile Robots.* *Artificial Intelligence*, 128(1-2):99–141, May 2001.

[111] T. Uthmann, C.Meyer, B. Schappel, and F. Schulz. *Team Characteristics: Mainz Rolling Brains 2000.* In W. van der Hoek, editor, *Proceedings of the European RoboCup Workshop*, May 2000.

[112] T. Uthmann, C.Meyer, B. Schappel, and F. Schulz. *Description of the Team Mainz Rolling Brains for the Simulation League of RoboCup 2000.* In P. Stone, T. Balch, and G. Kraetszchmar, editors, *RoboCup-2000: Robot Soccer World Cup IV*, pages 497–500. Springer Verlag, Berlin, 2001.

[113] T. Uthmann and D. Polani. *Between Teaching and Learning: Development of the Team 'Mainz Rolling Brains' for the Simulation League of RoboCup '99.* In *RoboCup-99 Team Descriptions for the Simulation League.* Linköping University Press, 1999.

[114] D. van Heesch. *The Documentation System Doxygen*, 1997. At http://www.doxygen.org.

[115] M. Veloso, P. Stone, and M. Bowling. *Anticipation: A Key for Collaboration in a Team of Agents.* In *Proceedings of the Third International Conference on Autonomous Agents*, 1999.

[116] A. Visser, J. Lagerberg, A. van Inge, L. Hertzberger, J. van Dam, A. Dev, L. Dorst, F. Groen, B. Kröse, and M. Wiering. *The Organization and Design of Autonomous Systems.* University of Amsterdam, Sept. 1999.

[117] N. Vlassis, B. Terwijn, and B. Kröse. *Auxiliary Particle Filter Robot Localization from High-Dimensional Sensor Observations.* In *Proceedings of the 2002 IEEE International Conference on Robotics and Automation (ICRA-02)*, Washington D.C., USA, May 2002. To appear.

[118] M. P. Wand. *Fast Computation of Multivariate Kernel Estimators.* *Journal of Computational and Graphical Statistics*, 3(4):433–445, 1994.