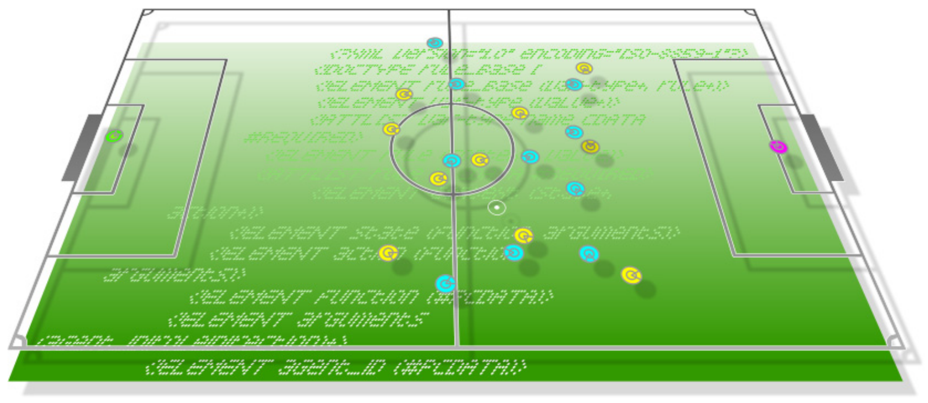
Using Coordination Graphs to add Prior Knowledge to the RoboCup Domain



Erik Burger

September 30. 2003

Using Coordination Graphs to add Prior Knowledge to the RoboCup Domain

Erik Burger

Supervisors:
Nikos Vlassis
Jelle R. Kok

Artificial Intelligence
Intelligent Autonomous Systems
Faculty of Science
University of Amsterdam

Keywords:
multiagent systems, coordination, coordination graphs, prior knowledge, RoboCup

The coverpage graphic was created by Martijn Burger, after a design by the author.

Acknowledgements

Although the actual writing of this thesis has been done by one person, there are a number of people without whom this thesis had never been written. These people, by offering support, ideas, critiques and various comments related and completely unrelated to the subject of this thesis, have guided the writing of this thesis as much as I did myself.

First and foremost, out of a tradition I have found to be completely justified, I would like to thank my supervisors Nikos and Jelle. Both have been invaluable during the course of my research, both providing an endless (and sometimes quite overwhelming) stream of ideas and comments. Without Jelle's expertise of the UvA Trilearn team at least half of my research would have been impossible to do. The other half would not have existed if not for Nikos. Nikos, if I did not often take up your offer of stopping by with questions whenever I wanted, the offer was highly appreciated, and the answers you provided when I *did* ask were always helpful.

A great amount of gratitude goes out to my family, for enduring me during the last year, for helping me get back to the right track after I had taken a wrong turn, but especially, for keeping my spirits up when things were not going as planned. Special thanks go to my mother, who was foremost in keeping me realistic with respect to deadlines (a lesson I will not quickly forget), and who has willingly subjected herself to countless readings and rereadings.

Finally there are a number of people who have also contributed, in their own way, to the writing of this thesis and my general well-being during this time. Nina, who did her utmost best to understand what I was doing and succeeded only partly, but was a great support nonetheless. Jan, who didn't decide to become a PhD student at the UvA to keep me company during and after lunches, but who did so anyway, providing me with much-needed breaks and useful commentary in the process. Matthijs, who was a great help in typesetting and formatting my thesis, and who had some valuable comments to add when he was not paying attention.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Multiagent Systems (MAS) is a relatively new research field in Artificial Intelligence (AI) that has been given increasing attention in both theory and practical applications [Vlassis, 2003]. MAS focusses on complex systems containing multiple, autonomous agents in which MAS aims to provide principles and theories for dealing with problems like coordination of behaviors and machine learning in multiagent domains.

An *agent* can be seen as anything (from an ant to a chess playing computer) that is situated in an *environment* and that *perceives* this environment though *sensors* and *acts* upon it through *effectors* [Russell and Norvig, 1995]. Effectors are anything that can be used by the agent to affect its environment. If the agent is an ant, its effectors would include its jaws and the gland that produces trail pheromones. For a chess playing robot, its effectors would consist of a pincer to pick up chess pieces, and various motors to control the robot arm. Besides this, an agent can have some knowledge of the environment, which can be used to reason about the possible actions available to the agent. Often, an agent has a goal which it tries to achieve through interaction with the environment or other agents. Each action will then bring the agent closer to its goal, or further away from it. This is often modelled by assigning *utilities* to actions; a value representing how good (or bad) a certain action is with respect to the goal the agent is trying to achieve.

When more than one agent is situated in the same environment we call this a *multiagent system*. In a multiagent system, an agent tries to model the behavior and goals of the other agents in its environment. This can be done either indirectly, through *observation*, or directly, through *communication*. Often a combination of the two is used.

In any situation, agents do not always share a common goal. Two ant populations might desire the same stock of food. In this case, where the goal of one group of agents is incompatible with the other, the other agents are seen as the group's adversaries, and the group will have to work together to a) maximize their own utility and b) minimize the other group's utility. Often, realizing the former will result in realizing the latter.

The use of MAS in the design of complex systems may have several advantages over using a single agent design. The most notable are *robustness*, *scalability*, *parallelism*, and *efficiency*.

Robustness is achieved by the fact that, when one agent in a multiagent system fails (either because it is unable to achieve its goal, or because it breaks down) the overall performance of the system will degrade gracefully. Up to a certain extent, other agents may take over the tasks left behind. In a single agent system, the failure of this agent will most likely result in a complete system breakdown.

Because agents in a MAS system are usually simple, small units, it is far easier to expand this system than it is to expand a single agent system, by adding new agents that may or may not have new capabilities available to them. This is called *scalability*.

Since there are several agents performing subtasks independently, a multiagent system is highly suitable for implementation using multiple processors or workstations. This inherent *parallelism* may not only increase the system's overall speed, but it also adds another layer of robustness.

Another important aspect of multiagent systems is *efficiency*. It is often far cheaper to design multiple agents whose behavior is not very complex than to design a single, highly complex system. Design times are shorter and debugging is easier and thus less time consuming. And time, as is commonly known, equals money spent.

One of the most exciting and dynamic domains for MAS research is that of robotic soccer as represented in the RoboCup Initiative. While a relatively new domain (the first RoboCup was held in 1998), it has quickly evolved to a large-scale platform for numerous areas of artificial intelligence and robotics research, amongst which are robot motor control, sensor fusion, strategy acquisition, and, more specific to the field of MAS, multiagent collaboration and coordination. The ultimate goal of RoboCup is to develop a team of humanoid robots that will be able to compete against human teams.

The initital objective of our research was to investigate real soccer situations and strategies and to see whether a mapping was possible from these situations to those within the RoboCup domain. This proved to be extremely difficult, as most situations in a real soccer match are the result of years of training and expertise, and even the slightest change in conditions might very well result in a vastly different situation. However, we found that there were indeed certain foundations to the decisions make by real soccer players that could be mapped into the decision making process of our robotic soccer players. As a result, we focussed our research on finding these fundamental rules.

We then considered various methods of implementing this prior knowledge into our robotic soccer team. Prior to our research, a framework called *coordination graphs* had been proposed by Guestrin [Guestrin et al., 2002a] as a means of eliminating the problem of exponential computational complexity in multiagent coordination, and a generic implementation of this framework had been implemented into our robotic soccer team [Kok et al., 2003].

The implementation of the CG algorithm used by our robotic soccer team, though it proved successful at the RoboCup World Championships in Padova, Italy, had several disadvantages. It was implemented to improve the passing behaviour of our soccer players, and thus was not very flexible with respect to inserting other types of rules. The passing rules were hard coded, so could not be changed without recompiling the soccer agent program, which made adapting the strategy of our soccer team during competitions tedious at best. In addition, the implementation was built to be fast enough to be used during RoboCup competitions, and although this surely was desired, it involved several obscurities that did not improve the clarity of the program code.

We decided to expand this implementation, aiming to create a system that was as flexible as possible with respect to inserting and changing the rules from our prior research, and fast enough to be useable in the various RoboCup competitions. As the original CG implementation involved only a small number of rules, we were curious to see whether the algorithm would still be usable in terms of speed with a greater number of rules. Answering this question was an important part of our experiments.

We further aimed for our implementation to be as clear as possible, allowing further development by others after the completion of our graduation project. The resulting implementation, as well as the results of the research that preceeded it, are presented in this thesis.

1.1 Guide to the Thesis

This thesis is organized as follows: in Chapter 2 we describe the various details of the RoboCup Initiative as the domain of our research. In Chapter 3 we present the results of our research of real soccer strategies and tactics, introducing the tactical system known as Total Football, and the movement system as one of its forms. Chapter 4 describes the coordination problem and introduces the notion of Coordination Graphs (CGs) as a means to represent the coordination requirements in a multiagent system. In Chapter 5 we present the extensions made to the UvA Trilearn Soccer Simulation Team as a result of our research, and we present a number of coordination rules describing a game of soccer and the implementation of these rules in the UvA Trilearn Team using CGs. In Chapter 6 we describe the experiments conducted to test our extensions and their results. We present our conclusions in Chapter 7, and we look ahead to possible angles and ideas for future research.

Chapter 2

The RoboCup Initiative

In this chapter we introduce the RoboCup initiative as the domain for our research. We begin with a description of the RoboCup's ultimate goal in Section 2.1, as well as its various short and longer term goals. Next we describe the RoboCup domain as the newest standard AI problem, comparing it to the already solved problem of computer chess, in Section 2.2. Section 2.3 continues with an overview of the organisation of RoboCup. In Section 2.4, we describe in detail the soccer server as the underlying framework of the RoboCup Simulation League, in which our research takes place, describing the soccer server program, the soccer monitor, and the logplayer, respectively. Finally, Section 2.5 describes the UvA Trilearn Soccer Simulation Team as another part of this framework.

2.1 Introduction

The Robot World Cup Initiative (RoboCup) 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 [Kitano et al., 1997]. The ultimate goal of RoboCup is 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."[Kitano and Asada, 1998]

This goal was proposed to be one of the grand challenges shared by the AI and robotics community for the next fifty years. Given the current state of affairs in AI and robotics this goal might sound overly ambitious, even unrealistic, and many people would claim that it may never be met. However, the challenge to build a computer chess program that could defeat the chess world champion encountered the same skepticism, and we have seen, in May 1997, that this challenge was met when IBM's Deep Blue was victorious over Gary Kasparov, beating him 3.5-2.5 over 6 games. Reaching this goal took only forty years. Also, it took only some 66 years from the first man-carrying powered aircraft built by the Wright brothers on the 17th of December 1903 to Neil Armstrong speaking the famous words “That’s one small step for [a] man, one giant leap for mankind” as he was the first man setting foot on the surface of the moon, on the 20th of July 1969.

What this clearly shows is that it is impossible to judge a challenge of a magnitude like that of the RoboCup as unrealistic. But to have to wait fifty years for the conclusion is a bit of a long time. Therefore various short-term goals have been proposed. Firstly, RoboCup is a project to promote robotics and AI research by providing a challenging problem. The tasks of RoboCup are varied and require a merging of various areas of robotics and AI, among which are design principles of autonomous agents, multi-agent collaboration, strategy acquisition, (real-time) sensor fusion, reactive behavior, learning, real-time planning, context recognition, motor control, strategic decision making, intelligent robot control, and many more [Kitano et al., 1997]. To reach RoboCup’s ultimate goal all these areas will need to be incorporated into a single working system and many breakthroughs in the various areas must be made. Even if the ultimate goal is not met, several advances will undoubtedly emerge from the effort to get there alone. This can be considered one of the short-term goals of the RoboCup initiative.

Another intention of the RoboCup organization is to use RoboCup for educational purposes and to stimulate the public’s interest in the fields of AI and robotics by offering a highly-dynamic, exciting, and appealing framework for research. Already, a large number of universities offer courses and study projects that are related to the different aspects of RoboCup. Additionally, the increasing number of publications concerning RoboCup other than technical papers reflect the growing interest from the media and the general public for the RoboCup competitions and related events.

2.2 RoboCup as a standard AI problem

Since the beginning of AI, standard problems have been the main driving force behind AI research. Research on computer chess, the most typical example of a standard problem, had led to several developments, among which are the discovery of some of the most powerful search algorithms to date. Other problems, like the Missionaries and Cannibals Problem (MCP)[1], have illustrated the difficulties involved in everyday reasoning. There are, however, several reasons why people might criticize the use of these problems. One of the reasons is that the problems are abstract ones, ignoring essential elements of the real world. While this is a valid reason for rejecting abstract problem solving tasks, it is often infeasible, socially, economically or for any other reason, for research institutes

[1]The MCP is usually stated as follows. Three missionaries and three cannibals are on one side of a river, along with a boat that can hold one or two. Find a way to get everyone across, without ever leaving the missionaries outnumbered by the cannibals on either side. [Russell and Norvig, 1995]

to focus on real world problems. Also, it is difficult to build rigorous theories on problems involving uncertain environments. It is therefore the aim of RoboCup to provide a standard problem which is both realistic and affordable, although the latter is, due to the participation of major companies like Philips and IBM in the Middle-Size league (described in Section 2.3), quickly changing. Universities are increasingly unable to keep up with the significant amounts of money these companies are able to spend on equipment.

As a sharp contrast to the standard problem of chess, RoboCup offers a dynamic, real-time environment, in which the agents are forced to deal with noisy, incomplete information. In addition to these difficulties, the information offered is non-symbolic and processing and control is done in a decentralized manner. Table 2.1 gives an overview of the different characteristics of the chess and RoboCup domains, respectively. With the achievement of the long-term objective of the computer chess problem, it is believed that the characteristics of the RoboCup domain, and more specifically the technologies that will emerge thereof, will be especially important for the next generation of industrial applications.

	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 2.1: Domain characteristics of computer chess compared to those of robotic soccer. From [Kitano and Asada, 1998].

2.3 RoboCup organization

The RoboCup competition contains a number of different leagues, each focusing on a specific part of the overall problem. This allows researchers to concentrate on several aspects of the problem individually, without having the overhead introduced by the other tasks. Currently, the following leagues are available:

- **Small Size Robot League (F-180).** In this league a team of five robots plays on a soccer field the size of an official-size table-tennis table. The robots are about 20cm high and have a diameter of about 15cm, and are colored to allow friend-foe identification. The ball is an orange golf ball. An overhead camera provides the a global view of the playing field. This view is sent to the controlling PC, which extracts the information needed and determines the actions to be executed by the robots. A game consists of two halves, each 10 minutes in length. Research areas which are important in this league include intelligent robot control, image processing and strategy acquisition.
- **Middle-Size Robot League (F-2000).** In this league a team may consist of no more than four robots, each about 75cm in height and with a

diameter of about 50cm. The dimensions of the playing field are approximately 9 by 5meters. The robots have no global information of the field as in the Small Size League. The matches last 20 minutes. Important research areas in this league include localization, vision, sensor fusion, robot motor control, and hardware issues.

- **Sony Legged Robot League.** In this league a team consists of four Sony quadruped robots (better known as AIBOs), including the goalkeeper. The playing field is of the same size as that of the Small Size League. Though the robots have no global information about the playing field, a number of colored markers is set up around the field to aid in robot localization. The length of a match is the same as in the Middle-Size League. The main problems addressed in this league are intelligent robot control[2] and the interpretation of sensory information[3].

- **Humanoid Robot League (H-40, H-80 and H-120).** The Humanoid League is the most recent addition to RoboCup. It was introduced in 2002, and has three sub-leagues. The differences in the three sub-leagues can be found in the size of the robots, with the height of the robot being the decisive factor. The heights are 40cm, 80cm and 120cm, respectively. All robots are biped, and competition events include walking a specified course and shooting penalties. Up until now the playing of a soccer match has been beyond the state of the art.

- **Simulation League.** The Simulation League is currently the largest of the five RoboCup Leagues, largely due to the fact that no expensive hardware is required to participate and teams can be easily tested against other teams. A team consists of 11 software agents which operate in a simulated environment, the *RoboCup Soccer Server*[Foroughi et al., 2001]. This system allows teams of agents to play against each other in a real-time environment. Sensing and acting are asynchronous, and various forms of uncertainty are added to the sensor inputs. Agents have limited perception of the playing field, communication is unreliable and low-bandwidth (meaning that only a limited number of characters can be transmitted at each interval), and agents are limited in their physical capabilities. The abstractions made in the *soccer server* allow researchers to focus more on strategy acquisition and learning than on movement and object recognition. The research areas explored in the Simulation League include machine learning, multiagent collaboration and opponent modeling, as well as the earlier mentioned strategy acquisition.

2.4 The RoboCup Soccer Server

The RoboCup Soccer Server is a soccer simulation system which allows teams of autonomous agents to play soccer matches against each other. It was originally

[2]An AIBO has as much as 20 degrees of freedom

[3]An AIBO has 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.

developed in 1993 by Dr. Itsuki Noda from the Electrotechnical Laboratory (ETL) in Japan. It has been used since the beginning of RoboCup as the basis for several competitions and research challenges.

The Soccer Server consists of three main components, namely the *soccer server* itself, the *soccer monitor* and the *logplayer*. These three components will be described globally in the following sections. For a more detailed description, see the thesis by Jelle Kok and Remco de Boer [de Boer and Kok, 2002], who provide an excellent source of information without becoming overly technical. More technical information can be found in the Soccer Server Manual [Foroughi et al., 2001].

2.4.1 The Soccer Server

The soccer server provides a domain (a virtual soccer field), simulates the movements of all objects on the field, and controls the soccer game according to several rules. It adds a certain amount of noise to the perception of the agents connected to it, regulates stamina loss and recovery, is able to simulate weather conditions (such as wind and rain) and provides a referee module, which is able to detect trivial situations like scoring, players in an offside position, and the ball going out of bounds. It is the aim of the soccer server to provide an environment that is as realistic as possible.

Agents (soccer players) can connect to the soccer server as clients, using a specified port. One program may only control one agent. Using the UDP/IP protocol the agents are able to send requests to the server to perform a certain action (like `kick` or `dash`). The server then processes these requests and updates the environment accordingly. The server also sends, at given intervals, information about the environment to the agents.

Though direct inter-agent communication is not allowed, the agents may use the `say` and `hear` commands to communicate indirectly. These commands are severely restricted in their use by several server-side parameters, like the length of a message (currently the maximal length of a message is 10 bytes) and the intervals at which the agents receive messages. This makes communication difficult and unreliable and thus advocates the use of MAS approaches that rely little, or not at all, on inter-agent communication.

The soccer server also provides and manages the various models of the agents and their environment, like the different *sensor models*. A soccer agent has three types of sensors: a visual sensor, a body sensor, and an aural sensor.

The *visual sensor* provides the agents with visual information of the playing field. The accuracy of this information as well as the intervals at which it is received can be changed by the agent program.

The *body sensor* reports physical information to the agent, such as its stamina level, speed and the angle of its neck. The information of this sensor is sent to the agent automatically at intervals defined by the soccer server.

Finally, the *aural sensor* enables the agent to hear other agents' messages, provided that whoever spoke the message is within hearing distance. An agent can only receive one message each time step, and if multiple messages are received within a certain time only the first is acknowledged. The other messages are discarded. This, too, makes communication very unreliable. The agent sensors are discussed in more detail in Section 3.2 of [de Boer and Kok, 2002].

The soccer server further provides a number of *action models*, which model the various actions an agent can perform. These actions are currently `kick`, `dash`, `turn`, `say`, `turn_neck`, `catch`, `move`, `change_view`, `sense_body`, `score` and more recently added, `tackle` and `point_to`. The agent action models are described in Section 3.4 of [de Boer and Kok, 2002], with the exception of the `tackle` and `point_to` models.

Lastly, the soccer server provides a *movement model*, which simulates the movements of all the objects on the field. It calculates the acceleration of an object, adds a noise vector to the movement of objects and models a wind vector, which in turn affects the movement of objects. The movement model is described in Section 3.3 of [de Boer and Kok, 2002].

First appearing in version 7 of the soccer server was the concept 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. Since soccer server version 7 however, each team can choose from several different player types with different characteristics. These player types are randomly generated, within limits, when the soccer server is started. In a match, both teams choose from the same set of player types. An example of a player type which is different from the standard type (which is still supplied, since not all teams make use of heterogeneous players) is a player which is faster than average, but gets tired more easily. The use of heterogeneous players makes the game more interesting. The problem of choosing where to put which playertype makes a (basic) knowledge of tactics indispensable.

2.4.2 The Soccer Monitor

The simulator includes a visualization tool, the *soccer monitor*, which allows researchers (and spectators) to see what is happening on the field during the course of a match. It is connected to the soccer server through UDP/IP. When the server and the monitor are connected, the server will send information about the state of the environment to the monitor in each cycle. The monitor then displays this information on a computer screen using the X Window system. A typical instance of the soccer monitor is shown in Figure 2.1.

The soccer monitor provides a two-dimensional, top-down view of the playing field. The concept of height therefore plays no role in the simulation. The dimensions of the field adhere to those in real soccer, except for the width of the goals. This is doubled due to the fact that it is harder to score in two dimensions than it is in three.
The players are shown as colored circles. Two lines originate from the center of each circle, indicating the direction the body of the player is facing and the

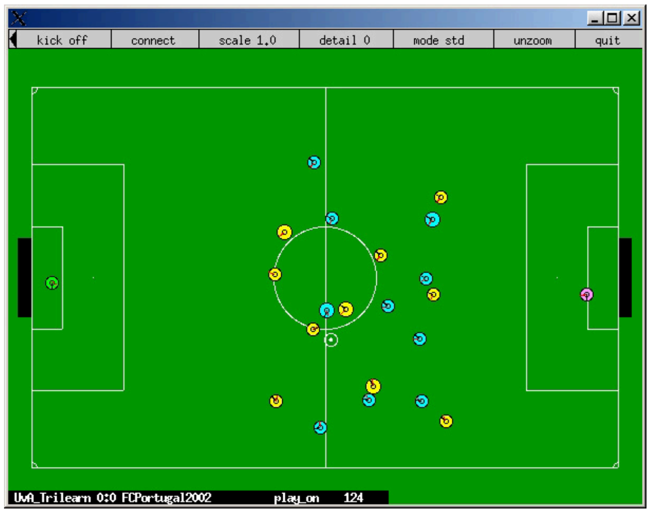


Figure 2.1: The soccer monitor display. The dimensions of the field are 105 by 68 meters, with each goal 14.02 meters wide.

direction the neck is turned to, respectively. Depending on the detail level, which can be set by clicking on the button labeled *detail* with either the right or the left mouse button[4], the players are labeled with no information at all, with their player number, their stamina information, or their player type. In the appropriate *view mode*, the view cone of an agent can be displayed as well, by pressing the number of the desired agent on the keyboard. The soccer monitor further displays the team names, the current score, the cycle number and the current play mode as determined by the referee module[5]. As the referee module is only able to discern trivial situations, a server interface is provided by the soccer monitor which allows a human referee to give free kicks to either team or to drop the ball on an arbitrary position on the field.

The tools provided by the soccer monitor are of great benefit to researchers, who can view the performance of their agents as they play. The ability to zoom into a part of the field and the ability to show more or less detail concerning the agents make debugging a lot easier, and thus more efficient. Another tool that provides functionality to simplify soccer agent development and debugging is the *logplayer*.

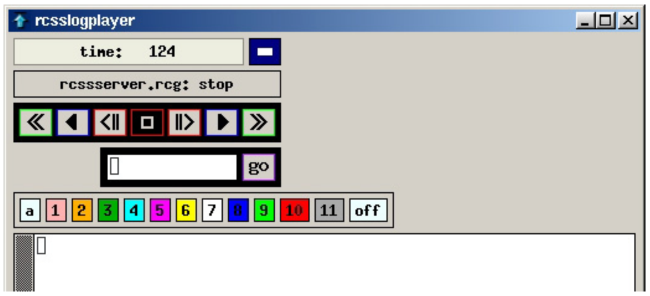


Figure 2.2: The logplayer. The colored buttons and the area below are an extension to the standard logplayer.

2.4.3 The Logplayer

The logplayer can be regarded as a video recorder which can be used to replay soccer matches. The soccer server can be started with the option of recording all current match data, storing this data on disk (in a *logfile*). The logplayer can then be used in combination with the soccer monitor to replay the recorded match. Just like a normal video recorder, the logplayer is equipped with *start*, *stop*, *fast forward* and *rewind* buttons. It can also show a match or situation in slow-motion. Another feature of the logplayer allows one to jump to a certain cycle of the match, which can be useful if you want to view only a certain part of the game, such as a goal.

In Figure 2.2 the logplayer is shown as it is used for the development of the UvA Trilearn soccer simulation team (see Section 2.5). Besides the buttons of the standard logplayer it features a row of colored buttons and a text area. These features are part of an extension called *layered disclosure*, first introduced by Peter Stone and colleagues in [Riley et al., 2000]. In this system, the developer defines an information hierarchy which may hold information about the agent's internal state and beliefs. The user may then request this information at any of the specified levels of detail, and either retroactively (i.e. through a logfile) or while the agent is acting. In this way, the user gains access to the reasons behind an agent's actions in addition to the world state as shown by the soccer monitor. Knowing the way the agent sees the world (which is, due to sensory noise and the dynamics of the environment, often different from the actual state of the environment) and the decision process that led to a certain action can be very helpful in debugging the agent program.

The layered disclosure system that was implemented in the UvA Trilearn team was built on top of the multi level log system that was already present in the team code. The buttons added to the standard logplayer allow the developers

[4] Clicking left will decrease the amount of details shown, while clicking right will increase the amount of detail.

[5] The most important play modes are the following: *before_kick_off*, *play_on*, *time_over*, *kick_off*_x, *kick_in*_x, *free_kick*_x, *corner_kick*_x, *goal_kick*_x, *goal*_x, *drop_ball*, *offside*_x, where x denotes the side to which the mode applies. It can be either l or r, indicating the right or left team. From [Foroughi et al., 2001]

to view the specific world model, observations and decision processes of the selected agent. This information is then displayed in the text area below the row of buttons. In addition to textual information it is also possible to view the agent's world model in a more graphical manner, using lines and circles drawn in the soccer monitor display. The combination of layered disclosure, the standard functions provided and the graphical display of the soccer monitor make the logplayer an extremely powerful debugging tool.

2.5 The UvA Trilearn Team

The UvA Trilearn soccer simulation team was built by two masters students, Remco de Boer and Jelle Kok, for their graduation project in 2001. The resulting team *UvA Trilearn 2001* was built from scratch, focusing mainly on the lower levels of the program and the basic skills of the players (shooting, passing, etc.). Finding the code of other teams lacking in both structure and documentation and considering the fact that, if they would decide to stop further development of the team, the code should be easy to read, understand and build upon, a lot of effort was spent on structuring the code. An object-oriented approach was decided upon, resulting in a program both highly accessible (the Object Oriented Programming (OOP) paradigm is one very well known and widely used) and easily scalable.

The development of the UvA Trilearn 2001 team, as mentioned, focused mainly on the lower levels of the agent program. Due to the fact that the code was build from scratch, a lot of improvement on the underlying code of other teams could be implemented. This, among others, led to an advanced synchronization scheme, effectively dealing with the synchronization problems introduced by the soccer server, and accurate estimation techniques for velocities and positions. The higher levels, added only after the lower levels were perfected to satisfaction, introduced an optimal scoring policy and a fast-play strategy using heterogeneous players. More information on the 2001 version of the UvA Trilearn team can be found in [de Boer et al., 2002] and [de Boer and Kok, 2002], and references therein.

UvA Trilearn 2002 [Kok et al., 2002a] was an extension on the 2001 team. The position and velocity estimation scheme was improved considerably by the use of particle filters. The agents' decision algorithm was improved upon as well, introducing a priority-confidence model [Lubbers and Spaans, 1998]. In this model, each agent assigns priorities to its different actions based on the agent's position on the field. Then a confidence value is calculated, indicating the chance that an action will succeed. The priority values and the confidence values are then combined to calculate a priority-confidence measure on the selected actions, after which the action with the highest value is selected for execution.

The UvA Trilearn team has proven to be very successful in the various RoboCup competitions. Currently it is two-year title holder in the German Open competition and current RoboCup World Champion. An overview of the team's results can be found in Table 2.2.

Competition	**Result**
German Open 2001	5th (of 12)
World Cup 2001	4th (of 42)
German Open 2002	**Champion** (of 13)
World Cup 2002	4th (of 42)
German Open 2003	**Champion** (of 12)
American Open 2003	**Champion** (of 15)
World Cup 2003	**Champion** (of 46)

Table 2.2: Results for the UvA Trilearn Team.

Chapter 3

Real Soccer Strategies and Tactics

In this chapter we present the results of our research of real soccer strategies and tactics. After a short introduction to the complexities of the different roles in a soccer game in Section 3.1, we describe the development of soccer tactics and the use of formations in Section 3.2. Next we introduce the tactical system known as Total Football in Section 3.3, starting with a description of its history and philosophy in Section 3.3.1. Then, in Section 3.3.2, we describe in more detail the movement system, one of the many forms of Total Football.

3.1 Introduction

"The object of soccer is to score more goals than the opposing team."

While the above is an extremely simple statement, it is, in essence, a complete description of the game. Unfortunately, actually playing the game of soccer is all but as simple as the above description would lead us to think. Ask a randomly chosen player, no matter the level of competition he (or she, of course) is in, what his task on the field is and you will receive a multitude of answers, few of which will be the simple 'to prevent the opposing team from scoring' or 'to score as many goals as possible'. The roles of players on the field differ, both from a tactical point of view as from what the crowd perceives happening. A defender's task is more than stopping the opposing team from scoring, while the tasks of the attackers do not merely exist of scoring as many goals as possible. Generally, what the supporter sees is but a glimpse of the real task a soccer player has to perform.

Ask a defender (also known as a *fullback*[1]) what his task on the field is, and ask him to elaborate. He will probably surprise you with the amount of subtasks he is required to perform to reach his (and his team's) ultimate goal: to end the match victorious. Some of the defender's subtasks are more obvious, others are more hidden. It is obvious to see a defender disrupt attack after attack. What is less obvious is the subgoal of this activity. The offense becomes agitated, their morale goes down, and subsequently they will be more prone to make errors. Another example of a defender's task is to provide a solid basis to the attack. By spreading out while in possession of the ball they force the opposing team to do the same, thus creating more space for the midfield and attack to maneuver in. By keeping possession when their own team is ahead, the defense wins valuable time in which the opposing team, obviously, cannot score goals.

The role of the attacker (or *forward*) is just as varied, and while the supporter watching from the sideline is generally satisfied by the forward scoring goals, the team's expectations are much higher. The forward, and this applies more specifically to the wing forward, is expected to be constantly on the move, drawing the opposing defense away from the path of the ball, facilitating the final scoring attempt. He is expected to fall back in times of danger, making the transfer from defense to offense easier and thus less risky. A forward should have great insight of the game, which is especially hard considering he is on the opposite end of the field, mostly facing the opposing team's goal. You might even extend the role of the forward to that of a defender. After all, in many formations and strategies defending begins as soon as the ball is lost, and after a failed attack, who is closest to the ball holder?

Remains the midfield, the most important part of a formation. The midfield is the spine of the team, the link between offense and defense. The midfield serves to slow an attack down so the defense can organize, and it provides the forwards with 'the right ball in the right place'. A team's strategy is often, if not always, build around its midfield, and often you can define a team's complete formation just by looking at the team's midfield [Glanville, 1979].

Clearly, the game of soccer is not as simple as it looks. This has not always been the case. In the next section I will give a brief overview of the changes in soccer tactics since the English 'invented' the game in 1863.[2] The official name of the game then was 'Association Football', which was later abbreviated to 'soccer'[3]. The name stuck.

[1]There appears to be some confusion concerning the terms *fullback*, *back*, and *halfback*. While the first two are one and the same, indicating a defender, the last is actually a midfielder. As other terms for defenders include *sweeper* and *stopper*, the confusion is quite understandable.

[2]Soccer as a game, of course, had existed for roughly nine centuries, then. It was in 1863, however, that the University of Cambridge bound the game to a set of rudimentary rules. In that same year the English Football Association (EFA) was formed.

[3]Inspired by the loose term for playing Rugby, which was known as playing 'rugger'.

3.2 A history of tactics

In the early stages of soccer tactics were largely absent. Dribbling was the way to play, and a typical match would consist of the player at the ball trying to pass as many defenders as possible, while the rest of the *six to eight* forwards followed closely behind, not as much to backup the ball holder but more in the hope that, when the ball would be lost, they would be the one recovering it. A player's skill was measured by his control of the ball and his ability to drive the opposing team's defenders crazy. Players chased the ball all over the field, leaving the goalkeeper as the only player on the field with a set position.

Defending was not an important issue back then, even though the development of tactics, up until the introduction of Total Football around 1970, was to focus more and more on the defense. With eight forwards in the 1860s, reduced to seven later that decade, there were only some four players left as a defense, including the goalkeeper, who was crippled all the more by not being allowed to use his hands until 1870[4]. The basic approach to defending was to clear the ball as soon and as far away from the own goal as possible, after which the cycle of attacking and defending began anew.

Things started to change in rapid succession after the Scots introduced the enlightened concept of passing in the beginning of the twentieth century. This caused a shift in formation. One of the six forwards was brought back to become a *center half*, creating a well-needed link between offense and defense. Of course the center half could not look after the midfield all by himself, and often the wing defenders or the inside forwards would reposition to assist him. Still the formation looked roughly as shown in Figure 3.1.

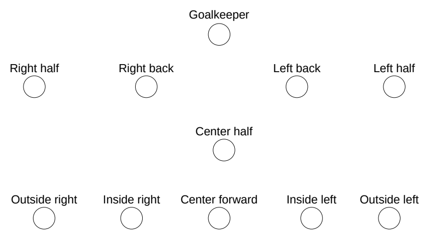


Figure 3.1: Common formation in the beginning of the 20th century.

Another shift occurred when the EFA, irritated by the abuse of the offside rule, changed that very rule, reducing the number of players to put a man onside from three to two. Consequently the center half was pulled back once again to function as a defender, later evolving into the *stopper*. To make up for the loss in midfield Arsenal invented the Third Back game, also termed the WM formation after the placement of the halfbacks and inside forwards. Figure 3.2 shows this formation. Note that, in fact, not much changed from the formation in Figure 3.1, except that the positions that were already commonly filled by the halfbacks and inside forwards were now made 'official'.

[4]A decision further refined in 1912, when the keeper's use of hands was restricted to his own penalty box.

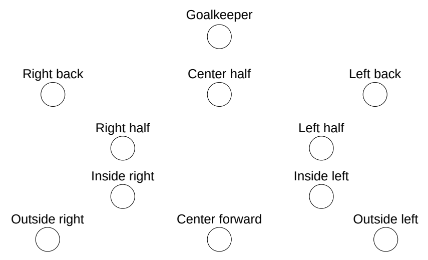


Figure 3.2: The Third Back game.

The Third Back game gave birth to a new defensive system known as *pivotal covering* (Figure 3.3). In this system, whenever the right flank was threatened, the right back would attempt to reclaim the ball. The central defender (still known as the center half, then) covered the right back, while the left back would cover the central defender. The same, of course, applied to the left flank. So, if the first defender failed to stop the opposing forward the next man in line would be there to solve the problem, and so on.

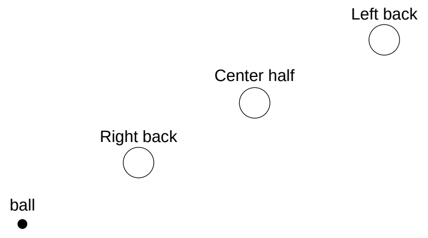


Figure 3.3: Pivotal covering.

It was this system of defense that caused the invention of 4-2-4[5], and later 4-3-3, by the Brazilian team. The Brazilians were unable to adopt pivotal covering into their playing style, much due to their wing backs, also called *volantes*. True to their name, they were an example of the classical half backs that existed before the Third Back game, moving upfield and downfield freely. This often left the central defender alone to defend his zone, which was quite impossible to do effectively. Instead of forcing their backs into the pivotal covering system, the Brazilians solved the problem with the same simple elegance they showed on the field. They added another defender, so that the central defender always had someone to back him up in times of need, and at the same time giving the *volantes* the freedom to move upfield, assured by the fact that there would always be a covering defender close at hand. The midfield was left to two players, assisted in turn by the backs and the wing forwards, while the four players at the front provided plenty strength to worry the opposing defense. The 4-2-4 formation, together with its successor, 4-3-3, is shown in Figure 3.4.

[5]Soccer formations are typically described as A-B-C, where A, B and C denote the number of defenders, midfielders and forwards, respectively. The 11th player is assumed to be the goalkeeper.

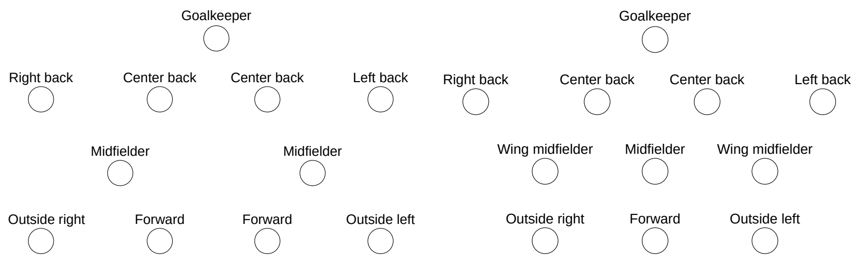


Figure 3.4: 4-2-4 and 4-3-3 formations.

The 4-3-3 formation was born more out of necessity than out of the intention to create a breakthrough in tactics. Other teams simply lacked the quality that made the two-man midfield of the Brazilians possible, and were forced to pull back a forward, thus bringing the count to three midfielders and three forwards. Brazil would eventually adopt this strategy as well, for as the team that originally introduced the 4-2-4 formation grew older, the midfield lost its speed and agility, and a change in tactics was required.

We have seen a tendency toward defense that was present since the beginning of tactics, and it was this tendency that created 4-4-2, which up to this day is one of the two most used formations, the other being 4-3-3. Both formations exist in a large number of variations, the most notable those with defensive systems using a *sweeper*[6], but the base has remained largely the same. For completeness, the schematic of the 4-4-2 formation is provided in Figure 3.5.

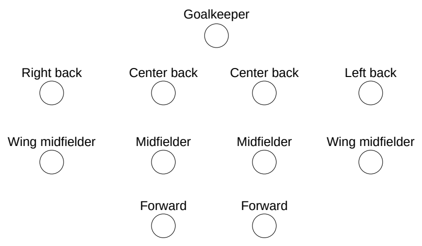


Figure 3.5: 4-4-2 formation.

The last great change in tactics up to today was the advent of Total Football. It is on this tactical system that we have based the larger part of our research, and we will describe it in more detail in the following section.

[6]Introduced by the ever-defensive Italians as a *libero*, the sweeper remains behind the three-man line of defenders, 'sweeping up' the balls that make their way past that line.

3.3 Total Football

3.3.1 History

The philosophy of Total Football was conceived as early as 1955, by the Viennese sports journalist and former goalkeeper of the Austrian national team Dr. Willy Meisl. In Soccer Revolution, a book by his hand, he spoke of a new concept in soccer he called *The Whirl*. In his idea, the future of tactics lay in fluidity, where players would abandon their normal positions and, like fluid, spread in all directions, typically into the opposing team's defensive half. To properly execute *The Whirl* a team would need players of immense skill and insight, jack-of-all-trades on the field, able to, at the whim of a moment, change their role from defender to midfielder to attacker, and back again. He believed that it would be possible to reach this breakthrough in soccer in as little as a few seasons. It turned out, though, that it would take the better part of two decades before Meisl's ideas would take root, personified in the teams of the Netherlands and West Germany.

In West Germany, it was Franz Beckenbauer that introduced his team Bayern Munich to what later would become known as Total Football. He was inspired by the Italian left back Fachetti, who had the habit of moving upfield and into the attack, adding to the successes of his club, Internationale. Beckenbauer reckoned that if a left back could use this tactic, then a right back could do as well, and why not the center of the defense? He further saw that the sweeper was in a sense an invisible player, hidden from view and consideration by the three defenders before him. That meant, according to Beckenbauer, that he could, if the opportunity arose, move from the shadows to assist in midfield, and even, if he was adventurous enough, in the attack. Beckenbauer's idea of an attacking sweeper proved successful, and the West German team proceeded to win both the European Cup in 1972 and the 1974 World Cup against The Netherlands. In addition to the attacking sweeper, the entire West German defensive line was often found participating in the attack, switching roles as easily as the all-round footballers Dr. Meisl had described twenty years earlier.

Meanwhile the Dutch, with major contributor Ajax, were building on what would be the greatest team in the history of Dutch football. Legendary players like Johnny Rep, Arie Haan, Wim van Hanegem and of course Johan Cruijff, due to the legalization of professional soccer in The Netherlands, no longer sought their fortune abroad and thus were free to be incorporated into the Dutch implementation of Total Football. The Netherlands, who had never been a football nation of any importance, quickly rose to the top of the world's soccer community, with two successive appearances in the World Cup Finals, in 1974 and 1978[7]. The development of Total Football in The Netherlands took place in parallel to the rise in Germany, and it was in Johan Cruijff that the Dutch found a Franz Beckenbauer of their own. Though it was the coach of the Dutch national team, the Romanian Kovacs, who implemented the concept, it was Cruijff who was seen to have made it a success. Of course, even he could not

[7]The Netherlands never managed to become World Champion; they lost 2-1 against West Germany in 1974, and 3-1 against Argentina in 1978.

have done it if the rest of the team had not consisted of such talented players.

Total Football, like the fluidity that was such a central concept, spread to all corners of the world, and nowadays most major soccer teams make use of the tactic, in one way or another.

3.3.2 A System of Play

Total Football, though the central philosophy has remained largely the same, can be found in many different forms. Some teams might only apply the concept to their offense and midfield, letting their defense adhere to the more traditional rules of formations and positions. Others, like the former team of West-Germany led by Franz Beckenbauer might have one or more players in a free role, adding then to the defense, then to the attack as the situation allows.

The *movement system* [Catlin, 1990] is one of the many forms of Total Football. It is appealing because of the simplicity of its philosophy, and because it is relatively easy to implement either partially or fully. The two central concepts of the movement system are *space* and *time*, both of which are natural implications of the concept of fluidity that lay at the base of Total Football. The following sections will describe the central concepts of the movement system and show how these concepts are related, and then continue to show how they can be influenced by the players on the field.

Of the two concepts of space and time, *space* is the more prominent one. A simple yet complete definition of space on a soccer field is the gaps between and around opponents, the areas on the field that hold no players of the opposing team. A player standing in such an area ‘is open’ or ‘has space’. Space allows a player to handle the ball unchallenged, to control it and pass it to another player unhindered. Space is used to maintain possession, and the ball can be advanced by making use of open spaces.

Space, if used correctly, is often distributed evenly across the field; more space is available at a distance from the ball, while around the ball players are closer together and space is more limited. When a team has more space, it is more difficult to defend against successfully, and it is harder for a team to score against a defense that allows only limited space.

Inseparable from the concept of space is the concept of *time*; time influences space, or the need for space, and space influences time. The more space a player has available, the more time he has to control the ball, look up, analyze the situation around him, and find a suitable player to pass to. The amount of time a player needs is dependent on the skill of the player, and when a player needs less time, he immediately needs a smaller amount of space. The amount of time a player has depends not only on the size of the space around him, but also on the speed and aggressiveness of the opposing team.

There are several ways in which a soccer team can influence space and time, all of which are in one way or another related to movement. The movement system defines three types of movement; movement related to support, movement after passing and movement to create or fill space. The difference between the three types of movement is not always clearly distinguishable, and often a player fulfills several of the types with one single movement.

Movement related to support, or support movement, is very important. A player that is well-supported will have a greater degree of confidence, and will as a result play better. A team that understands and makes good use of support movements will be hard to put out of possession. One of the major issues in support movement is the creation of passing options. Ideally, a player should be supported by three to four players. Two of these players provide passing options to the left and right of the player with the ball, while the other two provide backward and forward passing options, respectively. This gives the player with the ball four options to pass the ball, and the opposing team will have to cover all four. Even when one of the options is unavailable (e.g. when the forward passing option is not possible because of a possible offside situation), the remaining three options are very hard to cover. It is the task of the supporting players to make sure that the passing lanes between them and the player with the ball remain free. This allows the ball holder to concentrate on other things, like finding the best player to pass to, or deciding whether or not there is a possibility to dribble past an opposing player. Communication is an important part of support movement; often the player holding the ball will have his head down or turned in another way. Shouting is then the only way a supporting player can communicate his position.

When a player is unable to offer support (e.g. he is tightly marked by an opposing player), he should move away from the player with the ball. More often than not the opposing player marking him will move with him, leaving a space for another player to fill, thus making sure that the player with the ball is supported. This is an example of a movement that has two goals, both providing support and creating and filling space.

Movement after passing is a very common way to penetrate the opposing teams defense. After passing the ball the player moves downfield, in the direction of the goal, creating a forward passing option (or a through passing option). Moving after passing may trigger several penetrating passes, one of which is the *wall pass*. The wall pass is a very popular and effective way of passing defenders; the ball holder passes the ball to either his left or right, then sprints past his defender. The receiver of the pass then passes the ball directly back to the original passer. A great advantage of this pass is, beside getting the ball past a defender, that there is often a large amount of space behind a defender. This space then allows the team to advance still more.

Movement after passing can also be used to offer support, if needed. When a player passes the ball and the receiver of the pass has no passing options, the original passer should move to provide support. This can be done either by dropping back or by making a run into a space that provides a passing option. Lastly, movement after passing can be used to create space. A player making a run after passing the ball draws the attention of the defense, luring them into believing that the player is initiating a wall pass or through pass, or a similar penetrating movement. Thus the defender will move with the player, leaving behind space.

The final type of movement defined in the movement system is movement to create or fill space. We have already seen an example of this type of movement in the situation where a player who is in a support position is unable to give the support because of a tightly marking defender. As he moves away from his position he is *creating space* for an open team member. This team member then moves to *fill* the space created.

Movement to create space is not bounded to those players in a support position around the ball, or to the player in possession of the ball. It is used by all the players on the field, in offense, midfield, and defense. When a player makes a run forward after passing, he leaves behind a space to be filled by the nearest available player. When a defender engages an opponent, it is up to his fellow defenders to fill up the space that is left behind, thus not allowing an opposing player to make use of that space.

One of the most important ways to create space, and one of the most effective, is the use of ‘width’ and ‘depth’. By using width and depth, an offensive team can spread out across the entire length of the field. They consequently force the team on the defense to do the same, creating tremendous amounts of space. As said before, a team that has space is very difficult to defend against.

The opposite also applies. A defensive team will try to narrow the field of play, making the spaces available to the offensive team smaller, and thus making it easier to prevent them from penetrating the defense and score. Typically a team will continually switch from one end of the spectrum to the other, as ball possession is lost and regained.

Chapter 4

Coordination Graphs

In this chapter we introduce the notion of coordination graphs (CGs) as a means to represent the coordination requirements in a multiagent environment. We begin in Section 4.1 by describing the coordination problem and the problem of the exponential explosion of the joint action space in existing methods using Nash Equilibria as an example. Section 4.2 then introduces coordination graphs, with a description of the original CG algorithm in Section 4.2.1. Section 4.2.2 then describes two extensions to the original CG algorithm to overcome the limitations in using coordination graphs in the RoboCup domain.

4.1 Coordination Games

In a multiagent environment where all agents share a common goal (as is the case in the RoboCup domain), it is necessary that the agents work together to maximize the utility of their actions. Only in this way can they assure that their goal is reached in the most efficient manner. To accomplish this, the agents need to *coordinate* their actions. There are several methods for agents to coordinate their actions. One of the best known and attractively simple methods involves the selection of a Nash equilibrium from the set of available actions:

Let A_i be the set of actions available to agent i and $A = A_1 \times \ldots \times A_n$ the set of *joint actions* of all agents. Further let R_i be the payoff function for agent i such that $R_i(A) \rightarrow I\!R$ is a mapping from the joint action to a real value (i.e. the payoff for the selected joint action). A Nash equilibrium defines a joint action $a^* \in A$ with the property that for every agent i holds $R_i(a_i^*, a_{-i}^*) \geq R_i(a_i, a_{-i}^*)$ for all $a_i \in A_i$, where a_{-i} is the joint action for all agents excluding agent i. Such an equilibrium joint action is a steady state in which no agent can choose a different action and increase its utility given the actions of the other agents. Figure 4.1 shows a very simple situation in which two agents should coordinate their actions. The Nash equilibria are denoted in bold.

The problem apparent in using Nash equilibria is that the joint action space

	thriller	comedy
thriller	**1, 1**	0, 0
comedy	0, 0	**1, 1**

Figure 4.1: A simple coordination game: choosing a movie.

of a multiagent system is exponential in the number of agents present in the system. This means that, with a large number of agents, it becomes infeasible to determine the Nash equilibria of the system. A method to address this problem is the use of *context-specific coordination graphs*. This method has been recently extended to be better suited for the RoboCup domain [Kok et al., 2003, Kok et al., 2002b]. Before we discuss these extensions, we will first introduce the original method as described in [Guestrin et al., 2002b].

4.2 Coordination Graphs

A coordination graph (CG) represents the coordination requirements of a multiagent system. A node in the graph represents an agent, while the edges of the graph define a dependency between two agents: an agent i has a neighbor j if the action choice of agent j affects the payoff function of agent i. At any instance, only agents that are connected by edges need to coordinate their actions. These smaller coordination problems can be regarded as a decomposition of the global coordination problem, and the global payoff function is the sum of the payoff functions of the smaller problems. An example coordination graph for a system consisting of four agents is shown in Figure 4.2.

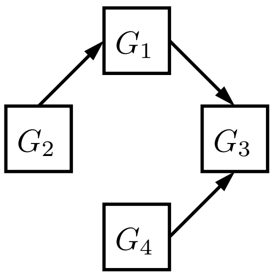


Figure 4.2: A CG for a 4-agent system.

In this figure agent G_1 has to coordinate with agents G_2 and G_3, agent G_2 has to coordinate with agent G_1, agent G_3 has to coordinate with agents G_1 and G_4, and agent G_4 has to coordinate with agent G_3. Using an effective variable elimination algorithm and a message passing scheme the agents can compute the optimal joint action for this instance.

4.2.1 The original CG algorithm

The CG algorithm is as follows; each agent is assumed to know its neighbors in the graph (but not necessarily their payoff function, which might depend on other agents). Each agent is eliminated from the graph by solving a local optimization problem that involves only the agents and its neighbors. The agent collects from its neighbors all relevant payoff functions (i.e. the functions that include the agent), then optimizes its decision conditionally on its neighbors' decisions. The agent then communicates the resulting conditional payoff function back to its neighbors. This process is then repeated for all agents, after which each agent, starting with the last agent eliminated and working back to the first, communicates its decision to its neighbors. They then use this decision to determine their own strategy, communicating this to their neighbors, and so on, until the cycle completes and the algorithm terminates, resulting in an optimal joint action.

In the context-specific CG algorithm the payoff functions are rule-based, as opposed to the matrix-based approach described in [Guestrin et al., 2002a]. In the rule-based approach the payoff function consists of a set of *value rules*, which specify how an agent's payoff depends on the current context. The context is defined as a propositional rule over the state variables and the actions of the agent and its neighbors. These rules can be regarded as a sparse representation of the complete payoff matrices.

As an example, consider the following situation taken from the pursuit domain (also known as the predator/prey domain) [Kok and Vlassis, 2003]: Two predators have cornered a prey and will be able to capture it in their next turn. The prey will be captured if one of the two predators moves onto the square containing the prey, but it will escape if *both* predators move onto the square containing the prey. Predators may not occupy the same square. This situation could be described by the following rule p_1:

$$\begin{array}{lll} \langle p_1 \;\; ; & \texttt{next-to-prey}(G_1) & \wedge \\ & \texttt{next-to-prey}(G_2) & \wedge \\ & G_1 = \texttt{move-onto-prey} & \wedge \\ & G_2 = \texttt{move-onto-prey} & : \;\; -100\rangle \end{array}$$

So, if both predators decide to move onto the prey, they will receive a utility of -100, which is assumed to lower the total utility enough to prevent this choice of action from occurring.

If in any other situation the state is not consistent with the above rule (e.g. one of the predators is *not* next to the prey), the rule does not apply and is discarded. By conditioning on the state and discarding all irrelevant rules the CG is dynamically updated and simplified. Thus, an agent only needs to observe that part of the state mentioned in its value rules.

For a more detailed example of how the CG algorithm works, consider the 4-agent coordination graph depicted in Figure 4.3(a). The value rules for each agent are given next to the agent, and the coordination dependencies are given by the directed edges between the agents. Note that in this example only binary states and actions are considered.

The first step of the algorithm is conditioning over the current context. After the agents observe $x = true$, the graph is conditioned, resulting in the removal of agent G_4 from the graph. This agent has no more value rules and thus does not affect the optimal joint action of the other three agents. The result of this step is shown in figure 4.3(b).

In the next step of the algorithm we begin the elimination process. We assume that the elimination order of the agents has been decided in advance, and agent G_3 is to be eliminated first. After collecting all relevant value rules from its child, agent G_1 (Figure 4.3(c)), agent G_3 has to maximize over the rules $\langle \overline{a_3} \wedge a_1 : 4\rangle\langle a_3 \wedge a_2 : 5\rangle$. For all possible action choices of agents G_1 and G_2, agent G_3 determines its best reponse, distributing the resulting conditional strategy $\langle a_2 : 5\rangle\langle \overline{a_2} \wedge a_1 : 4\rangle$ to its parent G_2 (Figure 4.3(d)). After this step, agent G_3 has no children in the coordination graph anymore and can be eliminated. The result is shown in Figure 4.3(e). Note that, due to the value rules communicated by agent G_3, one of agent G_2's rules is now dependent on the action choice of agent G_1, introducing a new child-parent relationship between these two agents.

The algorithm continues with agent G_2 collecting all relevant value rules from agent G_1. After maximizing over these rules, agent G_2 distributes the resulting conditional strategy to its parent G_1. Then agent G_2 is also eliminated, leaving only agent G_1 (Figures 4.3(f), 4.3(g) and 4.3(h)).

Agent G_1 then fixes its strategy to a_1. With no agents left to eliminate, the algorithm continues in the reverse elimination order. First agent G_1 communicates its decision to its children, agents G_2 and G_3. This step can be seen as the introduction of a new context variable, over which agents G_2 and G_3 condition their value rules (Figure 4.3(i)). Then agent G_2 fixes its strategy to $\overline{a_2}$ and communicates this decision to its child G_3 (Figure 4.3(j)). Finally, agent G_3 fixes its strategy to $\overline{a_3}$, resulting in the optimal joint action $\{a_1, \overline{a_2}, \overline{a_3}\}$, with a global payoff of 11.

4.2.2 Extensions to the CG algorithm

In applying this method to the RoboCup domain two limitations become apparent. Since the payoff functions consist of propositional rules, the algorithm requires a discrete domain. The RoboCup domain, however, is a continuous domain. Furthermore, in the course of the algorithm the agents are required to communicate twice; once to collect all relevant payoff functions from their neighbors, and again to distribute their conditional payoff function amongst their neighbors. As mentioned in the description of the soccer server (Section 2.4), communication in the RoboCup domain is severely limited and unreliable.

To be able to use the CG algorithm in a domain that is both continuous and where communication is unavailable, it is necessary to adapt the algorithm to deal with these domains. We will describe two extensions to the algorithm that

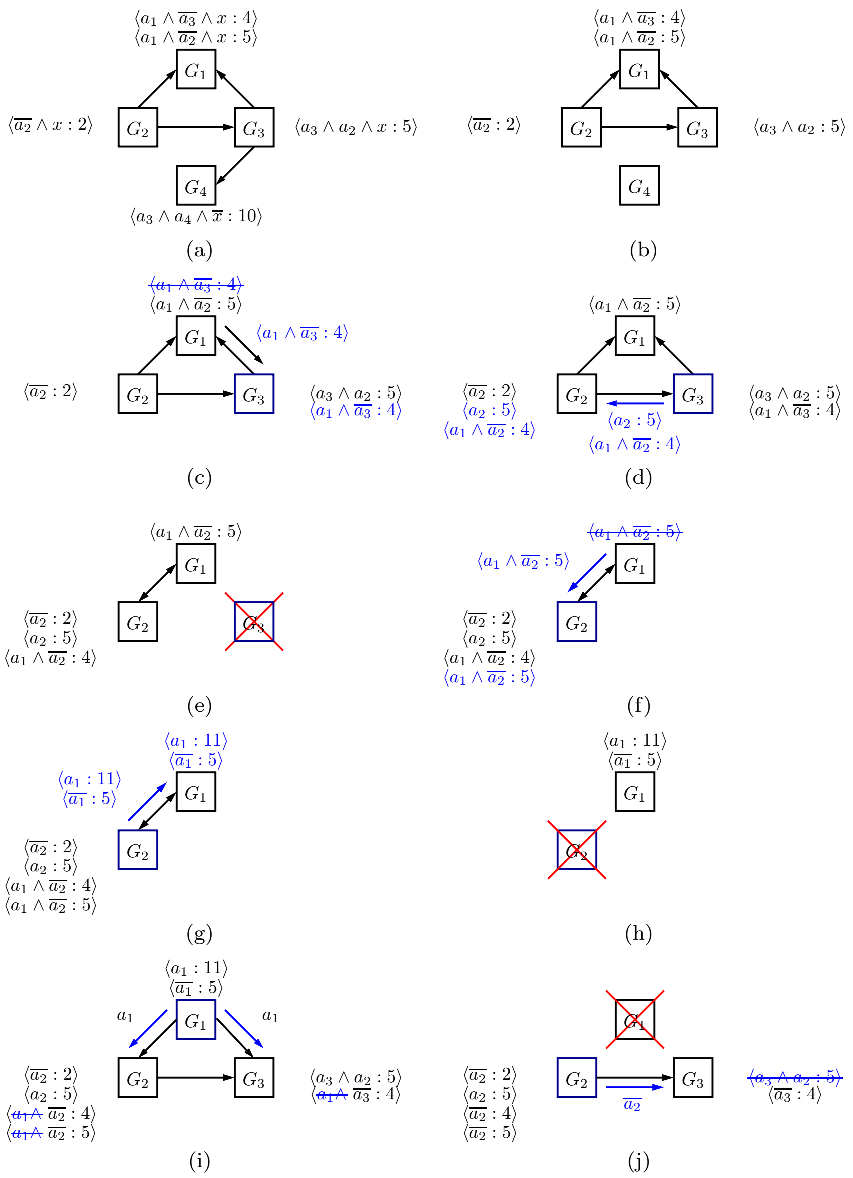

$\langle a_1 \wedge \overline{a_3} \wedge x : 4\rangle$
$\langle a_1 \wedge \overline{a_2} \wedge x : 5\rangle$
$\langle \overline{a_2} \wedge x : 2\rangle$
$\langle a_3 \wedge a_2 \wedge x : 5\rangle$
$\langle a_3 \wedge a_4 \wedge \overline{x} : 10\rangle$
G_1
G_2
G_3
G_4
(a)
$\langle a_1 \wedge \overline{a_3} : 4\rangle$
$\langle a_1 \wedge \overline{a_2} : 5\rangle$
$\langle \overline{a_2} : 2\rangle$
$\langle a_3 \wedge a_2 : 5\rangle$
(b)
$\cancel{\langle a_1 \wedge \overline{a_3} : 4\rangle}$
$\langle a_1 \wedge \overline{a_2} : 5\rangle$
$\langle a_1 \wedge \overline{a_3} : 4\rangle$
$\langle \overline{a_2} : 2\rangle$
$\langle a_3 \wedge a_2 : 5\rangle$
$\langle a_1 \wedge \overline{a_3} : 4\rangle$
(c)
$\langle a_1 \wedge \overline{a_2} : 5\rangle$
$\langle \overline{a_2} : 2\rangle$
$\langle a_2 : 5\rangle$
$\langle a_1 \wedge \overline{a_2} : 4\rangle$
$\langle a_2 : 5\rangle$
$\langle a_1 \wedge \overline{a_2} : 4\rangle$
$\langle a_3 \wedge a_2 : 5\rangle$
$\langle a_1 \wedge \overline{a_3} : 4\rangle$
(d)
$\langle a_1 \wedge \overline{a_2} : 5\rangle$
$\langle \overline{a_2} : 2\rangle$
$\langle a_2 : 5\rangle$
$\langle a_1 \wedge \overline{a_2} : 4\rangle$
(e)
$\cancel{\langle a_1 \wedge \overline{a_2} : 5\rangle}$
$\langle a_1 \wedge \overline{a_2} : 5\rangle$
$\langle \overline{a_2} : 2\rangle$
$\langle a_2 : 5\rangle$
$\langle a_1 \wedge \overline{a_2} : 4\rangle$
$\langle a_1 \wedge \overline{a_2} : 5\rangle$
(f)
$\langle a_1 : 11\rangle$
$\langle \overline{a_1} : 5\rangle$
$\langle a_1 : 11\rangle$
$\langle \overline{a_1} : 5\rangle$
$\langle \overline{a_2} : 2\rangle$
$\langle a_2 : 5\rangle$
$\langle a_1 \wedge \overline{a_2} : 4\rangle$
$\langle a_1 \wedge \overline{a_2} : 5\rangle$
(g)
$\langle a_1 : 11\rangle$
$\langle \overline{a_1} : 5\rangle$
(h)
$\langle a_1 : 11\rangle$
$\langle \overline{a_1} : 5\rangle$
a_1
a_1
$\langle \overline{a_2} : 2\rangle$
$\langle a_2 : 5\rangle$
$\langle \cancel{a_1 \wedge}\, \overline{a_2} : 4\rangle$
$\langle \cancel{a_1 \wedge}\, \overline{a_2} : 5\rangle$
$\langle a_3 \wedge a_2 : 5\rangle$
$\langle \cancel{a_1 \wedge}\, \overline{a_3} : 4\rangle$
(i)
$\langle \overline{a_2} : 2\rangle$
$\langle a_2 : 5\rangle$
$\langle \overline{a_2} : 4\rangle$
$\langle \overline{a_2} : 5\rangle$
$\overline{a_2}$
$\cancel{\langle a_3 \wedge a_2 : 5\rangle}$
$\langle \overline{a_3} : 4\rangle$
(j)

Figure 4.3: The CG algorithm in action.

allow it to be used in the RoboCup domain. The first extension is used to solve the problem of using the CG algorithm in a continuous domain. The second extension shows how the CG algorithm can be used when communication is not available.

It is difficult in the original CG domain to condition on a context that is defined over a continuous domain. A way to 'discretize' the context in the RoboCup domain is by assigning roles to agents [Spaan et al., 2002]. Roles are a natural and intuitive way of introducing domain-specific prior knowledge to the problem of distributing the global task of a team to its members. By using roles, one can create an abstraction from a continuous state to a discrete context, effectively introducing a discrete state variable (is an agent assigned a particular role or not?) depending on a continuous state variable. In the soccer domain, typical roles could include either general ones like *passive* or *active*, depending on for instance the distance to the ball, or more specialized roles like *forward*, *defender* or *goalkeeper*, depending on the player type or the agent's position on the field. The intuition is that, instead of coordinating the agents directly in a particular environment, we first assign roles to the agents based on the situation and then coordinate the set of roles. For more details on the assignment of roles to agents see [Spaan et al., 2002].

The use of roles can further reduce the action space by 'locking out' certain actions. For example, the role of the goalkeeper does not include the action `score`, nor does the role *forward* typically include the action `tackle`. This reduction of the action space offers computational savings and facilitates finding the solution of the local coordination problems. Consider the simple coordination game of Figure 4.1. If one of the agents is assigned a role that forbids him to choose the action `thriller` (e.g. because the agent is under 16), the other agent, assuming he knows how the roles are assigned, can safely choose the action `comedy`, resulting in coordination.

The second extension adresses the problem of non-communicating agents. When communication between agents is unavailable the CG algorithm can still be used under the requirement that the payoff function of an agent i is *common knowledge* among all agents that are connected to i in the coordination graph. Since only agents that are connected by edges need to coordinate their actions, this requirement frees agents from having to communicate their local payoff function during optimization. Furthermore, in the non-communicative case the elimination order of the agents neither has to be fixed in advance not has to be known among all agents. Each agent is free to choose an elimination order that allows the agent to compute its optimal action quickest. This is possible because the elimination order does not affect the resulting joint action, but only the speed of the algorithm.

The extended CG algorithm is roughly as follows (from [Kok et al., 2002b]):

> Each agent i maintains a pool of payoff functions, corresponding to all payoff functions of the agents in its subgraph. Starting from itself, agent i keeps eliminating agents until it computes its own optimal action unconditionally on the actions of other agents. For

each eliminated agent j the newly generated payoff functions are introduced into the pool of payoff functions of agent i and the process continues. In the worst case, agent i needs to eliminate all agents $j \neq i$ in its subgraph. Despite the fact that each agent computes its optimal action in a different way, the resulting joint action will always be the optimal one.

Clearly, the computational costs for each agent are increased to compensate for the unavailable communication. Instead of only optimizing for its own action, an agent (in the worst case) needs to calculate the optimal action for each other agent in its subgraph. The computational cost for each agent thus increases linearly with the number of payoff functions generated during elimination. However, the fact that communication is no longer necessary allows the different elimination processes to be run in parallel, while in the original CG algorithm the eliminations are performed sequentially.

A problem with this extension is that the assumption of common knowledge is strong and cannot be always guaranteed even when communication is available [Fagin et al., 1995]. In the non-communicating case common knowledge can only be guaranteed if all agents consistently observe the same world state. In a partially observable environment like the RoboCup domain this is not always easy to achieve.

To avoid this problem, it is required that, when the agents have to agree on a particular role distribution in a particular context, the parts of the state that are important in this distribution are, to a good approximation, fully observable by all agents. As coordination games in the RoboCup domain typically involve no more than two or three agents, this requirement is not too limiting and should be possible to achieve with relative ease.

Chapter 5

From Real Soccer to Robotic Soccer

In this chapter we apply the knowledge described in Chapter 3 to the UvA Trilearn Soccer Simulation Team. Using the existing architecture of the Trilearn Team, we first discern the changes made to the formation in Section 5.1. Next we specify value rules as introduced in Chapter 4 that represent the game of soccer in Section 5.2. Finally we give a description of the implementation of the coordination graph framework in Section 5.3 that makes use of the value rules.

5.1 Formations

The use of formations within a soccer team allows players to define their positions relative to each other in a way that is best suited for a certain situation or opponent. As we have seen in Section 3.2, different formations developed as a result of changes in the way soccer was played and because of changes in the rules of the game (e.g. the introduction of the offside rule). A formation determines whether a team plays offensively or defensively, and whether a team relies more on long passes downfield or short passes across the width of the field. A team's formation, too, may determine how much space a team has available.

In the UvA Trilearn Team, a formation is defined by both player types and player positions [Reis and Lau, 2001]. There are eight pre-defined player types: *goalkeeper*, *central defender*, *sweeper*[1], *wing defender*, *central midfielder*, *wing midfielder*, *wing attacker* and *central attacker*. Using a configuration file each player on the field is assigned a player type and a home position. This home position is used to calculate the players' *strategic position* on the field during play. Other parameters include a maximal and a minimal x-coordinate, defining the zone in which a player may move, and so-called *attraction factors*, i.e. how

[1]The *sweeper* player type was not implemented in the original UvA Trilearn Team and was added as a result of our research.

much the player is ‘pulled towards the ball’, or how much the ball’s position influences a player’s strategic position. Lastly, a parameter is provided to define whether or not a player should always attempt to stay behind the ball. The specification of the 4-3-3 formation currently used by the UvA Trilearn Team is shown in Table 5.1. The resulting formation is illustrated in Figure 5.1.

	player number										
	1	2	3	4	5	6	7	8	9	10	11
`home_x`	-50.0	-16.5	-21.0	-15.0	-16.5	0.0	0.0	-3.0	15.0	18.0	18.0
`home_y`	0.0	10.0	0.0	0.0	-10.0	-11.0	11.0	0.5	-0.5	19.0	-19.0
`pl_type`[2]	1	4	3	2	4	6	6	5	8	7	7

	player type								
	0	1	2	3	4	5	6	7	8
`attr_x`	0.0	0.1	0.7	0.65	0.7	0.65	0.7	0.5	0.6
`attr_y`	0.0	0.1	0.25	0.4	0.25	0.3	0.25	0.3	0.25
`beh_ball`	0	1	1	1	0	0	0	0	0
`min_x`	0.0	-50.5	-42.0	-47.0	-45.0	-36.0	-36.0	-2.0	-2.0
`max_x`	0.0	-30.0	0.0	7.0	7.0	42.0	42.0	44.0	44.0
`extra_y`	0.0	0.0	0.0	0.0	5.0	0.0	4.0	1.0	0.0

Table 5.1: Specification of the 4-3-3 formation currently used by the UvA Trilearn Team.

In this table, a home position is defined for each player by `home_x` and `home_y`. These variables indicate a player’s initial position, unaffected by the position of opponent players or the position of the ball. The center of the playing field has coordinates (0,0), and all home positions are relative to those coordinates. The player type of each player is indicated by the variable `pl_type`. Note that more than one player can have the same player type. This is due to the fact that, in most cases, a team’s formation is symmetrical, i.e there is a right and left wing defender, a right and left wing midfielder, and a right and left attacker (for illustration, see the formations depicted in Section 3.2). The roles of these symmetrical positions are so much alike that the choice of not defining separate player types is justified.

For each player type, a number of variables is defined such that players of the same type use the same values. This, again, can be justified by looking at the symmetry of common formations. To prevent a player from wandering too far from its home position, disrupting the team’s formation, `min_x` and `max_x` define a minimal and a maximal horizontal player position, respectively. These bounds also assure that a player does not waste stamina on running too far up- or downfield. The variable `beh_ball` is a boolean to assure that a player stays behind the ball at all times. This is, naturally, especially useful for defenders. `Attr_x` and `attr_y` are factors that define how much the position of the ball influences a player’s strategic position. For example, a player that aggressively ‘hunts’ the ball would have high attraction factors, while a defender that ‘waits’ for the ball to come to him would have a low value assigned to `attr_x`, but a

[2]The player type numbers denote the following types: 1=*goalkeeper*, 2=*central defender*, 3=*sweeper*, 4=*wing defender*, 5=*central midfielder*, 6=*wing midfielder*, 7=*wing attacker* and 8=*central attacker*.

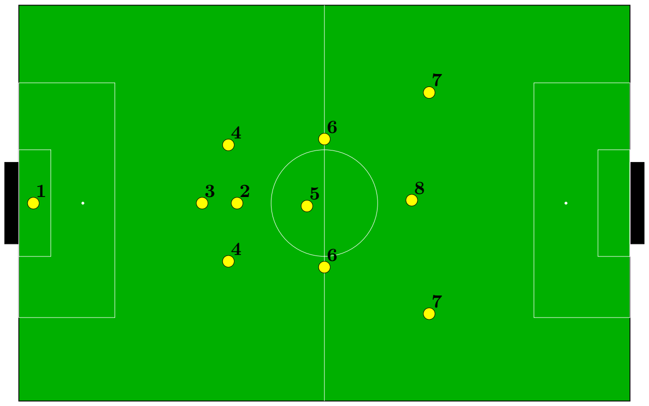


Figure 5.1: The 4-3-3 formation currently used by the UvA Trilearn Team as defined by Table 5.1. The player types are shown in bold.

---

higher value assigned to `attr_y`, thus ensuring that he is in the path of the ball when it arrives.

We have extended the specification of the formation with an extra parameter, `extra_y`. This variable represents the idea of adding 'width' to the playing field when in possession of the ball, as follows; the value of `extra_y` is added to a player's strategic position, so that that player moves to a position closer to the sideline. This has two results: the opposing team, if it wants to mark those players spreading out, will have to spread out as well, creating space. Also, the distances between players increase, preventing players from bunching together, and thus making passes longer and more effective. When the team is not in possession of the ball, the value of `extra_y` is ignored, effectively creating the effect of the team narrowing the playing field. This, as described in Section 3.3.2, causes the space available to the opposing team to be smaller, making it easier to organize the defense.

Another extension to the formation system of the UvA Trilearn Team was the introduction of the *sweeper* player type. As briefly mentioned at the end of section 3.2, the sweeper was introduced by the Italians as a one-man last line of defense, supporting the three defenders in front of him. The role of the sweeper is to intercept any ball that might pass the first line of defenders, as well as filling a defender's position temporarily if that defender moves to engage an attacking player. Using a sweeper it becomes increasingly hard for the opposing team to penetrate the defense. *Through passes*, where the ball is passed into the space behind the defenders, after which the receiving player runs into that space and thus penetrates the defense, are made close to impossible by the sweeper, and a player managing to dribble past one defender finds himself facing the sweeper next.

But as we have seen in Section 3.3.1, the role of the sweeper is not necessarily limited to that of a defender. Often unmarked, the sweeper can move forward to assist in midfield, or even in the attack. Due to the limitations on stamina of our agents, we were unable to implement this literally. However, we simulate the advance of the sweeper to the attack in the UvA Trilearn Team by switching formations on gaining ball possession. In addition to the 4-3-3 formation, we specified a 3-3-4 formation. When switching to this formation, the central defender moves to play in midfield, and the central midfielder joins the forwards, giving us an extra player in the attack. This numerical advantage is especially useful against teams with a tight or crowded (i.e. having four or more players) defense.

5.2 Coordination Rules

To be able to use the CG algorithm to coordinate our agents, we need value rules. Using the knowledge from Chapter 3, we are able to define a number of rules that make up a soccer game. Recall that, in our discussion of coordination graphs in Chapter 4, we limited ourselves to binary rules. For ease of implementation, we do the same in defining the rules we will be discussing. There are a number of limitations when converting real soccer rules into robotic soccer rules. Before we begin describing the rules we created, we first mention these limitations, and give a solution to them where possible.

5.2.1 Limitations

The first limitation we encounter in trying to create rules that are suitable for implementation in a soccer simulation team is *communication*. In every soccer book there is an emphasis on communication. This, of course, makes sense: when a player is free to receive the ball, but is unsure whether his teammate knows he is, he yells out his position. When two players plan to use the wall pass to pass an opponent player, they will have to communicate to make their intentions known. To do this, they do not necessarily have to shout. Using body language is an equally important part of communication on the soccer field. By turning their head or body in a certain direction or by simply pointing to where they want to receive the ball, players are able to derive each other's intentions.

As we have seen, communication in the Soccer Server is extremely limited. Thus agents cannot revert to yelling their intent, and body language is as of yet virtually unusable (agents simply cannot spare the time it takes to turn in a certain direction just to give an indication of their intentions). Recently, the `point_to` command was added to the Soccer Server, and it's use might prove beneficial in the future. Sofar, though, it is highly underutilized. The solution to the absence of communication was given in Section 4.2.2, in our discussion of the extensions to the original CG algorithm. By making an agent's payoff function common knowledge amongst all agents connected to it in the coordination graph, those agents are able to 'know' the agent's intent, doing the same mindreading as real soccer players.

Another limitation is the frequent use of *height* in real soccer tactics. One of the best-used tactics for advancing the ball quickly toward the opponent goal is using the *long pass*. The long pass involves passing the ball over the heads of the opponent team's midfield, often passing directly from a defender to a forward. The same idea applies to the *wide pass*, a pass from one side of the field to the other. It is used mostly to disorganize an opponent's midfield or defense by changing the field of play. A third example of a team using height to its advantage is corner kicks or free kicks. A free kick or corner kick is often taken by passing the ball high into the opponent team's goal area, where a teammate may header the ball into the goal.

In the Soccer Server, height does not play a role, and thus we are cut off from using long or wide passes. Passing from player to player to achieve the same effect is possible, but less effective. Beside the introduction of a third dimension in the next version of the Soccer Server there is no ready solution for this problem, but it is one that all teams have to deal with. So, despite the fact that our rules become somewhat less representative of a real soccer game, we simply ignore all tactics involving long or wide passes, or using headers.

5.2.2 Definitions

In constructing our rules we defined the following variables:

- *agent* is defined as the agent executing the CG algorithm.
- t is defined as an enumeration of all teammates of *agent*. Thus, when *agent* denotes agent 4, t will have values 1, 2, 3, and 5 to 11, respectively. When more than one t is used in a rule (e.g. t_1, t_2), it is assumed they denote different teammates.
- o is defined as an enumeration of all opponents of *agent*. When more than one o is used in a rule (e.g. o_1, o_2), it is assumed that they are not equal.
- *dir* is a variable denoting a direction. It can have one of the following values: *center*, *north*, *north_west*, *west*, *south_west*, *south*, *south_east*, *east* and *north_east*. In this case, when more than one *dir* is used, it is not necessarily assumed that the two are not equal.
- *goal* is the position of the own goal.

We also defined the following roles:

- *active_defender*. This is a specialist role. An agent is an *active_defender* if he is the closest defender to the ball, and an opponent player is in possession of the ball.
- *passive_defender*. Another specialist role. If one defender is assigned the role *active_defender*, the other defenders are assigned the role *passive_defender*.
- *sweeper*. Only one player can be assigned this role. If defender roles (*active_defender*, *passive_defender*) have been assigned, this role is assigned to the player having the *sweeper* player type.

- *interceptor*. An agent has the role *interceptor* if he is the fastest player to the ball, but the ball is not within kicking distance.
- *passer*. An agent has this role when he is in possession of the ball.
- *receiver*. An agent is a *receiver* if he is within passing distance of the agent closest to or in possession of the ball.
- *passive*. This role is assigned to all agents that are not assigned one of the above roles.

These roles are assigned in the order in which they are given here. Specialized roles are thus assigned before more general roles. Besides these variables and roles, we defined a number of *state variables*. These variables map to functions that return either true or false, depending on the current context. We will discuss these as they are introduced in our rules. We further defined a set of *actions*. These, too, will be discussed as they appear in our rules.

What is important to note is the use of *uninstantiated variables* in our rule descriptions. For example, *dir* is an uninstantiated variable that can take the values mentioned above. The use of uninstantiated variables allows us to define rules that may apply to more than one agent and to more than one situation. During the instantiation phase of our CG implementation all rules with uninstantiated variables are expanded into their instantiated counterparts (e.g. a rule with a *dir* variable will result in nine rules, each with a different value for *dir*). An added advantage is that the rulebase is smaller and less complex, allowing us to quickly change a rule or add new rules.

5.2.3 Rules

Our first set of rules involves *passing*. Passing successfully is extremely important. Every failed pass results in the loss of ball possession, and each loss of possession may potentially result in a goal by the opposing team. Using the rules for passing has an advantage beyond creating successful passes. By running the CG algorithm, a receiving agent knows at which position relative to himself the pass will arrive, *before* the pass is actually made. Originally, an agent knew nothing about the pass until he noticed the ball moving in his direction. Thus, knowing beforehand where the pass will arrive, the agent can begin moving earlier compared to the original situation. This gives our team a distinct advantage over opponent teams.

Consider the following rule p_1:

$$\begin{aligned}\langle p_1^{passer} \quad ; \quad & \texttt{has-role}(t, receiver) \quad \wedge \\ & \texttt{not-pass-blocked}(agent, t, dir) \quad \wedge \\ & \texttt{is-empty-space}(t, dir) \quad \wedge \\ & \texttt{not-offside}(t, dir) \quad \wedge \\ & a_{agent} = \texttt{pass-to}(t, dir) \quad \wedge \\ & a_t = \texttt{move-to}(dir) \quad : \quad u(agent, t, dir)\rangle\end{aligned}$$

Before we explain what this rule means, we will first explain how our rules are constructed in general. First, using the principle of roles as described in Section 4.2.2, we indicate the role this value rule will belong to in the superscript of the rule name. In this case, rule p_1 belongs to the role *passer*. In our implementation, thus assigning rules to specific roles allows for faster elimination due to the fact that certain roles might have only very little rules assigned to them (e.g. the role *passive*, in our implementation, is assigned only one rule). Then we assign the other roles that will be active in the rule. In this case we assign the role *receiver* to t. Note that we treat the role assignments as binary functions, the same as we did the state variables. This makes mapping our rules to the actual implementation easier. Next comes a list of state variables: `not-pass-blocked`($agent$, t, dir), that determines whether a pass from *agent* to the direction *dir* relative to t is blocked by an opponent or not, `is-empty-space`(t, dir), that returns whether the direction *dir* from t is free from opponents, and `not-offside`(t, dir), that returns whether moving in the direction *dir* will put agent t in an offside position. Then we assign actions to the different agents involved. In this case, agent t should move in direction *dir*, and *agent* should pass to the direction *dir* relative to t. Finally we pass the relevant parameters to a utility function.

It is not very useful to give constant utility values to rules whose utility may depend on changing factors (e.g. a player's position on the field). For example, an agent passing backward relative to itself may yield a lower utility when the agent is in his own defending area to reflect the higher risk involved. If we would have to write down every rule with a utility for each possible situation our rulebase would become enormous and far more complex. Therefore, we decided to calculate our utilities in real-time, i.e. for each cycle of a match. The variables passed to u are used to calculate these utilities. In the case of passing (rules p_1 to p_5), the utility depends on the position on the field of the *passer* and the *receiver*, and the direction in which the pass is made. Generally, passing forward (i.e. in the directions *north*, *north_west*, and *north_east*, as well as to a *receiver* with an x-coordinate closer to the opponent goal) will yield a higher utility than passing backward. A possible exception might be the situation where both *passer* and *receiver* are in front of the goal. In this case, passing across the width of the field (i.e. in the directions *east* and *west*) can serve to disorganize the defense, and will yield a higher utility than passing forward.

What rule p_1 means is relatively straightforward. From the perspective of the agent executing the CG algorithm, it means the following: "If I am in possession of the ball, and one of my teammates is close enough to receive a pass, and that pass is not blocked by an opponent, and my teammate has room in the direction I will be passing, and my teammate will not be offside if he moves in that direction, then I pass to my teammate, while he moves to the direction I am passing to."

Rule p_2 makes use of the idea that a player can 'know' what his teammate will do after running the CG algorithm:

$$\begin{aligned} \langle p_2^{receiver} \quad ; \quad & \texttt{has-role}(t, interceptor) \quad \wedge \\ & \texttt{not-pass-blocked}(t, agent, dir) \quad \wedge \end{aligned}$$

$$\begin{aligned}
&\texttt{is-empty-space}(agent, dir) \quad \wedge \\
&\texttt{not-offside}(agent, dir) \quad \wedge \\
&a_{agent} = \texttt{move-to}(dir) \quad : \quad u(agent, t, dir)\rangle
\end{aligned}$$

Using rule p_2, the agent moves in a direction his teammate could pass to when that teammate has intercepted the ball. Thus this rule is an example of *support movement*, as we described in Section 3.3.2. The state variables in rule p_2 are largely the same as those in rule p_1: the pass from t to *agent* should not be blocked by an opponent, *agent* should not move into an offside position, and direction *dir* from *agent* should be free from opponents.

Another example of support movement is when the receiver that is not being passed to as a result of rule p_1 moves to a supporting position:

$$\begin{aligned}
\langle p_3^{receiver} \quad ; \quad &\texttt{has-role}(t, passer) \quad \wedge \\
&\texttt{not-pass-blocked}(t, agent, dir) \quad \wedge \\
&\texttt{is-empty-space}(agent, dir) \quad \wedge \\
&\texttt{not-offside}(agent, dir) \quad \wedge \\
&a_{agent} = \texttt{move-to}(dir) \quad : \quad u(agent, t, dir)\rangle
\end{aligned}$$

This rule is almost identical to rule p_2, but it deals with the situation where the *interceptor* has intercepted the ball, and thus has become a *passer*. Should this player decide to dribble rather than pass, the receiver will move with him to provide a passing option.

We can also make more complicated rules. Rule p_4 involves three agents:

$$\begin{aligned}
\langle p_4^{receiver} \quad ; \quad &\texttt{has-role}(t_1, passer) \quad \wedge \\
&\texttt{has-role}(t_2, receiver) \quad \wedge \\
&\texttt{not-pass-blocked}(t_2, agent, dir_1) \quad \wedge \\
&a_{t_1} = \texttt{pass-to}(t_2, dir_2) \quad \wedge \\
&a_{t_2} = \texttt{move-to}(dir_2) \quad \wedge \\
&a_{agent} = \texttt{move-to}(dir_1) \quad : \quad u(agent, t_2, dir_1)\rangle
\end{aligned}$$

In this rule, *agent* will move to the direction that t_2 will (possibly) be passing to after t_2 has received the pass from t_1.

Thusfar our passing rules are very general. We can easily create rules for specific situations, like when two agents are in front of the goal and the agent in possession of the ball needs to decide whether to keep the ball or to pass it to his teammate. Rule p_5 represents this situation:

$$\begin{aligned}
\langle p_5^{passer} \quad ; \quad &\texttt{has-role}(t, receiver) \quad \wedge \\
&\texttt{in-front-of-goal}(agent) \quad \wedge \\
&\texttt{in-front-of-goal}(t) \quad \wedge \\
&\texttt{has-better-chance-of-scoring}(t, agent) \quad \wedge \\
&\texttt{not-pass-blocked}(agent, t, dir) \quad \wedge \\
&a_{agent} = \texttt{pass-to}(t, dir) \quad : \quad u(agent, t, dir)\rangle
\end{aligned}$$

There are two new state variables in this rule: `in-front-of-goal`(*agent*) returns true if *agent* is in front of the goal, i.e. in the opposing team's goal area, and `has-better-chance-of-scoring`(t, *agent*) compares the chances of scoring of agents t and *agent*, returning true if agent t has a better chance of scoring than *agent*. This rule states that if a teammate has a better chance of scoring than the agent in possession of the ball, and the pass is not blocked, then the agent should pass the ball. If no teammate has a better chance of scoring, this rule will not apply and another course of action will be taken (like the agent trying to score himself).

Our next set of rules describes *defending*. In defending it is very important that all defenders coordinate their actions such that any defender, at any given time, is backed by a teammate. Rules p_6 and p_7 define the actions available to the defender closest to the opponent in possession of the ball:

$$\begin{aligned}\langle p_6^{active\_defender} \quad ; \quad & \texttt{has-ball}(o) \quad \wedge \\ & \texttt{can-tackle}(agent, o) \quad \wedge \\ & a_{agent} = \texttt{tackle}(o) \quad : \quad u(agent, o)\rangle\end{aligned}$$

When the agent is close enough to the opponent in ball possession to attempt a tackle, then he should attempt to tackle. We introduce two new state variables in this rule: `has-ball`(o) determines whether o is in possession of the ball, and `can-tackle`(*agent*, o) returns true if *agent* is close enough to o to attempt to tackle. The action `tackle`(o) should speak for itself.

If the agent is not close enough to the opponent to attempt a tackle, rule p_6 does not apply. Rule p_7 states that in this situation the agent should position himself between the opponent and the goal:

$$\begin{aligned}\langle p_7^{active\_defender} \quad ; \quad & \texttt{has-ball}(o) \quad \wedge \\ & a_{agent} = \texttt{mark-line}(o, goal) \quad : \quad u(agent, o)\rangle\end{aligned}$$

The new action `mark-line`(o, *goal*) causes the agent to take a position along the line between the opponent o and the goal.

The next rule determines the action for a passive defender:

$$\begin{aligned}\langle p_8^{passive\_defender} \quad ; \quad & \texttt{has-ball}(o_1) \quad \wedge \\ & \texttt{marks}(agent, o_2) \quad \wedge \\ & \texttt{not-pass-blocked}(o_1, o_2, center) \quad \wedge \\ & \texttt{not-offside}(o_2) \quad \wedge \\ & a_{agent} = \texttt{mark-line}(o_2, o_1) \quad : \quad u(agent, o_1, o_2)\rangle\end{aligned}$$

When a pass is possible between the opponent in possession of the ball (o_1) and the opponent *agent* is marking (o_2), then *agent* should move to a position along the line between o_2 and o_1, preventing the pass. Note that in the state variable `not-pass-blocked`(o_1, o_2, *center*), the *dir* variable is instantiated to *center*.

This is because we assume that our opponent does not use (the same set of) CG rules to determine their actions. Therefor we cannot assume that o_2 will move to the direction o_1 would be passing to if he *did* use our CG rules.

In every other situation, it should suffice that the defender moves to his strategic position:

$$\begin{aligned}\langle p_9^{passive\_defender} \quad ; \quad & \\ & a_{agent} = \texttt{move-to-strategic-position}() \quad : \quad u(agent)\rangle\end{aligned}$$

The sweeper should cover the other defenders if they move to engage an opponent player:

$$\begin{aligned}\langle p_{10}^{sweeper} \quad ; \quad & \texttt{has-role}(t, active\_defender) \quad \wedge \\ & \texttt{has-ball}(o) \quad \wedge \\ & a_t = \texttt{tackle}(o) \quad \wedge \\ & a_{agent} = \texttt{mark-line}(goal, o) \quad : \quad u(agent, o)\rangle\end{aligned}$$

And move to his strategic position otherwise:

$$\begin{aligned}\langle p_{11}^{sweeper} \quad ; \quad & \\ & a_{agent} = \texttt{move-to-strategic-position}() \quad : \quad u(agent)\rangle\end{aligned}$$

Our last set of rules describes the default actions for each role, as well as deal with some very simple situations. Rule p_{12} states that if the agent in possession of the ball has space in direction *dir*, he should dribble in that direction:

$$\begin{aligned}\langle p_{12}^{passer} \quad ; \quad & \texttt{is-empty-space}(agent, dir) \quad \wedge \\ & a_{agent} = \texttt{dribble-to}(dir) \quad : \quad u(agent, dir)\rangle\end{aligned}$$

And when the agent in possession of the ball is in front of the goal, he should try to score:

$$\begin{aligned}\langle p_{13}^{passer} \quad ; \quad & \texttt{is-in-front-of-goal}(agent) \quad \wedge \\ & a_{agent} = \texttt{score}() \quad : \quad u(agent)\rangle\end{aligned}$$

Note that when another rule yields a greater utility (e.g. when another agent has a better chance of scoring (rule p_5), or a pass is possible (rule p_1), the agent will chose the action resulting from that rule instead of trying to score.

Rule p_{14} represents the situation where the agent in possession of the ball has a chance of beating a defender by dribbling past him:

$$\begin{aligned}\langle p_{14}^{passer} \quad ; \quad & \texttt{is-engaging-defender}(o) \quad \wedge \\ & \texttt{has-chance-to-dribble-past}(agent, o, dir) \quad \wedge \\ & a_{agent} = \texttt{dribble-to}(dir) \quad : \quad u(agent, o, dir)\rangle\end{aligned}$$

We introduce two new state variables: `is-engaging-defender`(o) is somewhat like assigning a role to an opponent player. A player is an *engaging defender* if he is close to the player in ball possession, and trying (or going to try) to take the ball from that player. `Has-chance-to-dribble-past`$(agent, o, dir)$ returns true if the chance that *agent* can beat the defender o by dribbling in direction *dir* is above a pre-defined threshold.

The roles *interceptor* and *passive* have only one action choice. If the agent is an *interceptor*, he should intercept the ball (i.e. try to get ball possession):

$$\langle p_{15}^{interceptor} \quad ; \quad a_{agent} = \texttt{intercept}() \quad : \quad u(agent)\rangle$$

If the agent is *passive*, he should move to his strategic position:

$$\langle p_{16}^{passive} \quad ; \quad a_{agent} = \texttt{move-to-strategic-position}() \quad : \quad u(agent)\rangle$$

5.3 Implementation

In implementing the CG algorithm and our soccer rules to enable the UvA Trilearn Team to make use of them, we decided upon the following two qualities we felt were desirable in such an implementation:

- *Speed.* In the Soccer Server, one cycle of a match has a length of 100 milliseconds. In these 100 milliseconds, all agents have to update their world model, determine their next action, and finally send their actions back to the Soccer Server. The execution of the CG algorithm should take place during the phase in which the agent determines his next action. At times, due to the asynchronous implementation of the Soccer Server, an agent determines his action twice per cycle. From the experiments performed in Chapter 5 of [de Boer and Kok, 2002] it shows that the CG algorithm has less than 8 milliseconds to complete[3].
- *Flexibility.* We wanted it to be easy to change an existing rule, or add new rules, without having to recompile our agent program. This would allow us to change the strategy of our team depending on the opponent team during competitions by either changing the rulebase or loading an entirely different one. Also, we wanted it to be (relatively) simple to extend the set of state variables, or to add new actions. Since the state variables and actions map to functions within the agent program, having to recompile the program after adding either one of these was inevitable.

[3]The experiments in Chapter 5 of [de Boer and Kok, 2002] were performed on an AMD Athlon 700Mhz machine with 512MB of RAM.

5.3.1 Rules

To achieve the level of flexibility we desired, we decided to code our rules using XML [Morrison, 2001]. XML, or eXtensible Markup Language, is rapidly becoming the industry standard for the encoding and communication of data. It allows the creation of custom *tags* that describe the content held between these tags. For example, the XML line

```
<color>red</color>
```

would describe *red* as being a *color*. Tags start with '`<`' and end with '`>`'. A tag starting with a forward slash ('`/`') is called an end tag, indicating the end of the content. Each start tag must have an end tag.

One of the characteristics of XML is that documents created in XML must adhere to a very strict structure. This structure is defined in the header of the documents, in what is called the Document Type Definition, or DTD. The DTD describes the elements (the tags) that appear in the document and their relation. For example:

```
<!ELEMENT soccerteam (player+)>
 <!ATTLIST soccerteam name CDATA #REQUIRED>
  <!ELEMENT player (#PCDATA)>
```

The above DTD defines a *soccerteam* as consisting of one or more (the '`+`' sign) *players*. `ATTLIST` describes the attributes for our *soccerteam*. In this case, it has the attribute *name*, which holds the name of the team, as `CDATA` (regular text). `#REQUIRED` makes sure that the *name* attribute cannot be omitted. It then defines a *player* as `#PCDATA`. `#PCDATA` stands for Parsed Character Data, which has the same meaning as `CDATA`, except that the first is considered content and the latter is considered as the value of an attribute. In the case of a player, `#PCDATA` could be the shirt number of a player, or a player's name, or anything else we feel is descriptive enough. Using the above DTD, we can create a document describing the Dutch national soccer team in 1970, as follows:

```
<soccerteam name="Dutch National Team">
  <player>Johan Cruijff</player>
  <player>Wim van Hanegem</player>
  <player>Johnny Rep</player>
    .
    .
</soccerteam>
```

The above example shows another advantage of XML: *readability*. Without knowing the DTD of the document, almost anyone that reads the above example can surmise what the meaning of the various tags and their content is. This is part of the reason why XML is so well suited for the encoding of data. For our purposes, yet another advantage is that a great number of generic XML parsers exist. These parsers run on most different platforms, so researchers may use the platform of their choice (e.g. Windows or Linux) for their implementation.

```
      <!DOCTYPE rule_base [
        <!ELEMENT rule_base (var-type+, rule+)>
          <!ELEMENT var-type (value+)>
           <!ATTLIST var-type name CDATA #REQUIRED>
        <!ELEMENT rule (context, value)>
         <!ATTLIST rule ID CDATA #REQUIRED>
 <!ELEMENT context (state+, action+)>
            <!ELEMENT state (function, arguments)>
            <!ELEMENT action (function, arguments)>
              <!ELEMENT function (#PCDATA)>
              <!ELEMENT arguments (agent_id|role|direction)+>
                <!ELEMENT agent_id (#PCDATA)>
                <!ELEMENT role (#PCDATA)>
                <!ELEMENT direction (#PCDATA)>
          <!ELEMENT value (#PCDATA)>
      ]>
```

Figure 5.2: The Document Type Definition for our soccer rules.

Figure 5.2 shows the full DTD for our soccer rules. The `DOCTYPE` tag defines a name for our DTD, in this case *rule_base*. Then we define our tags. Our outermost tag is *rule_base*, which consists of one or more elements of type *var-type* and one or more *rule*s. The *var-type* elements define the possible values for each variable type. The name of this type (e.g. *role*, see Section 5.2.2) is an attribute as defined in `ATTLIST`. The possible values for each variable type are defined by a number of *value* tags, which are defined at the bottom of our DTD. Figure 5.3 shows the definition of the *role* variable type.

```
<var-type name="role">
  <value>interceptor</value>
  <value>passer</value>
  <value>receiver</value>
  <value>passive</value>
</var-type>
```

Figure 5.3: Definition of the *role* variable type.

We then define our *rules* to contain a *context* and a *value*. The *context*, in turn, contains both *state* and *action* tags. These map to the state variables and actions as described in Section 5.2.3. The identifier (ID) for each *rule* is defined as an attribute. These identifiers are the same as the numbers of the rules in Section 5.2.3. The definitions of the *state* and *action* tags are identical: both contain a *function* (e.g. `has-role` or `move-to`) and its *arguments*. The *arguments* are defined by their type, which is one of *agent_id*, *role*, or *direction*.

Figure 5.4 shows an example definition of a state variable in which the role *passer* is assigned to agent *j*.

```
<state>
  <function>has-role</function>
  <arguments>
    <agent_id>j</agent_id>
    <role>passer</role>
  </arguments>
</state>
```

Figure 5.4: Assigning the role *passer* to agent *j* as a state variable.

5.3.2 Classes

A characteristic of XML is the ease in which tags map to C++ classes. To store the information from our XML document in a way that can be used in the CG elimination algorithm, we have defined a number of relatively simple classes that map closely to the tags we defined in our XML document's DTD. We defined the following classes:

- *Argument* represents the parameters as passed to a state variable or action. It contains a *value* and a *type* variable indicating the variable type of *value* (i.e. one of *agent_id*, *role* or *direction*).

- *State* represents the state variables and actions of our rules. The class has four memeber variables: *type*, which indicates whether the *State* variable is an action or a state variable, *function*, which holds the name of the function (e.g. `move-to` or `has-role`), *stateIsTrue*, which allows us to define negated state variables (e.g. `not-offside`), and a vector of *Argument* variables, the parameters of *function*. It also contains two member functions, *makeTrilearnContext* and *makeTrilearnAction*, which return a context or an action in a format that can be used by the CG elimination algorithm discussed in Section 5.3.4.

- *Context* is defined as a vector of *State* variables to represent the fact that it may contain one or more state variables and actions. It contains a member function *addState* to add a state variable or action to the context. It also contains a member variable, *instMap*, which is used by another member function, *instantiate*. This function is used to generate contexts that contain only instantiated variables from a context with uninstantiated variables.

  InstMap holds each variable in the context, along with its possible values. When adding a variable to *instMap*, it is checked whether the variable is instantiated. If it is, then it is added to *instMap* with itself as its only possible value. If it is not instantiated, it is added to *instMap*, and all its possible values are read from *domainMap*, a member variable of the

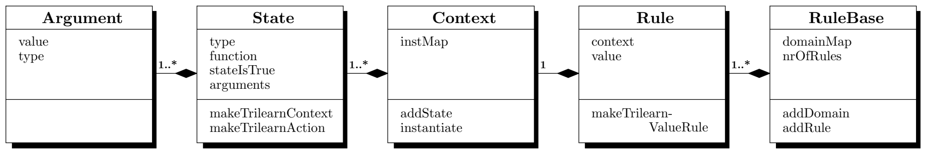


Figure 5.5: UML diagram of the classes defined for the XML parser.

RuleBase class. For example, *north* is an instantiated variable, and thus is added to *instMap* with itself as its possible value. *Dir* is uninstantiated, so it is added to *instMap* with all nine directions (see Section 5.2.2) as its possible values. When called, the *instantiate* function will loop through *instMap*, creating a context for every possible combination of values, thus resulting in contexts that hold only uninstantiated variables.

- *Rule* represents the coordination rules we defined in our XML document. Similar to the definition in the Document Type Definition, the *Rule* class contains two member variables: a *Context* and a *value*. Also, it contains a function *makeTrilearnValueRule*, that calls the *makeTrilearnContext* and *makeTrilearnAction* member functions of *State* to map the content of a *Rule* to a format that can be used by the CG elimination algorithm.

- *RuleBase* is our main class. It is defined as a vector of *Rule* variables. Besides a number of functions to show the curent content of the rulebase it contains two member variables: *nrOfRules* holds the number of rules currently in the rulebase, and *domainMap* holds all variable types and their possible values. This information is read from the XML tags *var-type* in the XML document. As discussed above, *domainMap* is used to generate contexts (and thus rules) that contain no uninstantiated variables.

  The first member function of *RuleBase*, *addDomain*, handles the setting of the *domainMap* variable. The second member function, *addRule*, takes two parameters, a *Context* (or a vector of *Context* variables) anda *value*, and adds these to the rulebase as a new rule. Consequently, the *nrOfRules* variable is updated to reflect the new number of rules in the rulebase.

Figure 5.5 shows the *Unified Modeling Language* (UML) diagram for our classes. UML is a language for specifying, visualizing, constructing and documenting the artifacts of a software system [Rumbaugh et al., 1999]. UML supports nine different types of graphical diagrams, one of which is the *class diagram* used in Figure 5.5. Classes are represented by rectangles which are divided into three rows. The top row contains the name of the class, the middle row contains a list of attributes for the class (the member variables), and the bottom row contains the operations defined for these attributes (the member functions). Relationships between classes are indicated by different kinds of arrows which indicate the kind of relation. In Figure 5.5 the only relation used is the *composition* relation. This means that one class is part of another (e.g. the *Rule* class is part of the *RuleBase* class, since a rulebase consists of one or more rules.). Multiplicity notations are placed near the arrows. These indicate the number of instances of one class linked to one instance of another class. For example, one *RuleBase* may contain one or more (1..*) instances of *Rule*.

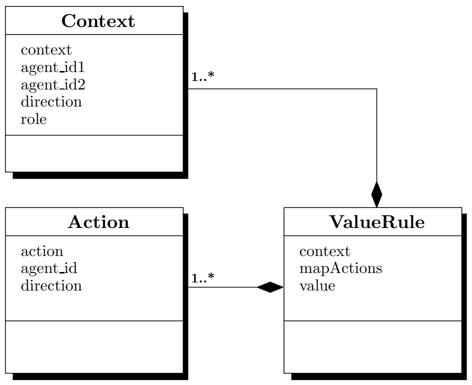


Figure 5.6: UML diagram of the classes defined in the UvA Trilearn Team's implementation of the CG algorithm.

5.3.3 Parser

To parse our XML rules we used the *Expat*[4] parser. We chose to use this parser because it has built-in support under Linux. Expat is a so-called *non-validating* parser, which means that it does not check whether the XML rules presented to it are well-formed with respect to the Document Type Definition. This allows for faster parsing.

Expat parses XML using an *event-driven* approach. This means that, while reading an XML document, certain situations (e.g. the reading of an end tag) will trigger the calling of the function linked to that event. Commonly, there are three events that are required to be handled by the appropiate function: the reading of a starting element[5], the reading of an end element, and the reading of character data. In our implementation, these events were handled by the *startElement*, *endElement*, and *charElement* functions, respectively. The pseudo code for the XML parser is depicted in Algorithm 5.1.

- *startElement* handles the processing of start tags. It's main tasks are setting the *currentTag* variable to the current tag being processed and processing the attributes of that tag, if any. Depending on the current tag, it performs the appropiate action.
- *endElement* handles the processing of end tags. Depending on the value of the *currentTag* variable, it stores the data initialised by the *startElement* and *charElement* functions in the appropiate data structures (as described in Section 5.3.2), thus building up the rulebase for use by our agents.
- *charElement* basically handles everything else. It processes and stores the names of functions and variables to be later on processed by the *endElement* function.

[4]http://www.jclark.com/xml/expat.html
[5]Expat uses the term *element* instead of *tag*.

State variables	
`has-role`	CONTEXT_HAS_ROLE
`not-pass-blocked`	CONTEXT_PASS_NOT_BLOCKED
`is-empty-space`	CONTEXT_EMPTY_SPACE
`in-front-of-goal`	CONTEXT_IN_FRONT_OF_GOAL
`not-offside`	CONTEXT_NOT_OFFSIDE
Actions	
`move-to`	COORDACT_MOVE_*dir*[6] COORDACT_HOLD
`pass-to`	COORDACT_PASS
`move-to-strategic-position`	COORDACT_GOTO_STRAT_POS
`intercept`	COORDACT_INTERCEPT
`dribble-to`	COORDACT_DRIBBLE
`score`	COORDACT_SCORE

Table 5.2: The possible state variables and actions and their mapping to Trilearn contexts and actions. The left column shows the state variables and actions defined in our implementation, the right column shows the corresponding contexts and actions used by the UvA Trilearn Team.

5.3.4 Mapping

Our implementation of the XML parser and the classes used focussed on clarity rather than speed. Parsing the XML rules and building the rulebase is done only once, in the beginning of the agents' program, and thus speed was of little importance.

However, the generic CG algorithm that was used by the UvA Trilearn Team during the RoboCup World Championships in Padova was built to be fast, which resulted in the use of a different class architecture [Kok et al., 2003]. Therefore it was necessary to create a mapping from our implementation as discussed in Sections 5.3.1, 5.3.2, and 5.3.3 to the implementation used by the Trilearn Team. This resulted in the *makeTrilearnValueRule* function, which in turn calls the *makeTrilearnContext* and *makeTrilearnAction* functions. The pseudo code for these three functions is given in Algorithm 5.2.

The UvA Trilearn Team's implementation of the CG algorithm defines three classes. The UML diagram for these classes is shown in Figure 5.6.

- *ValueRule* represents a rule in the coordination graph, and contains a *value*, a vector of *Context* variables, and an array *mapActions* which contains an *Action* for each agent involved in the rule.
- *Context* defines a state variable and its parameters. The possible state variables are shown in Table 5.2. An example context might look like this:

[6] *Dir* denotes one of the following directions: NORTH, NORTHEAST, EAST, SOUTHEAST, SOUTH, SOUTHWEST, WEST and NORTHWEST. The direction *CENTER* results in COORDACT_HOLD.

Context(CONTEXT_HAS_ROLE, 1, -1, DIR_ILLEGAL,
ROLE_PASSER)

In this context agent 1 has role *passer*. The third and fourth parameters (a second agent and a direction) are not used and are set to -1 and DIR_ILLEGAL. Compare this to Figure 5.4, which describes the same context but with j uninstantiated.

- *Action* defines the possible actions for an agent and its parameters. The possible actions are shown in Table 5.2. An example action might look like this:

  Action(COORDACT_PASS, 4, DIR_NORTH)

  The agent executing this action will pass to agent 4 in the direction *north*.

```
global tmp
global currentTag = the tag being processed
global rulebase = the rulebase to be created

{ startElement function }
set currentTag to the current tag being processed
if currentTag == var-type then
   add the value of the name attribute to tmp
else if currentTag == state then
   clear tmp
else
   continue processing
end if

{ endElement function }
local context = the context of the rule being processed

if currentTag == var-type then
   { tmp contains the variable name and its possible values }
   add tmp to the domain map of rulebase
   clear tmp
else if currentTag == state then
   { tmp contains the function name and its arguments }
   add the state defined by tmp to context
   clear tmp
else if currentTag == action then
   { tmp contains the action name and its arguments }
   add the action defined by tmp to context
   clear tmp
else if currentTag == rule then
   { tmp contains the rule's value }
   { context contains the uninstantiated context of the rule }
   call instantiate on context
   add the resulting instantiated context and tmp to rulebase as a new rule
   clear tmp
   clear context
else
   continue processing
end if

{ charElement function }
local str = the string being processed

if currentTag == value, function, agent_id, role, or direction then
   add str to tmp
else
   continue processing
end if
```
Algorithm 5.1: Pseudo code for the XML parser.

```
{ $makeTrilearnValueRule$ function }
local $rule$ = the rule being processed
local $trilearnValueRule$ = a rule in the format of the CG algorithm
local $trilearnContext$ = the context used to create $trilearnValueRule$
local $trilearnActions$ = the actions used to create $trilearnValueRule$

for all $states$ in $rule.context$ do
  if $state.type$ == "state" then
    call $state.makeTrilearnContext$
    add the resulting state variable to $trilearnContext$
  else if $state.type$ == "action" then
    call $state.makeTrilearnAction$
    add the resulting action to $trilearnAction$
  end if
end for
create $trilearnValueRule$
return $trilearnValueRule$

{ $makeTrilearnContext$ function }
create $trilearnContext$ depending on $state.function$ (see Table 5.2)
return $trilearnContext$

{ $makeTrilearnAction$ function }
create $trilearnAction$ depending on $state.function$ (see Table 5.2)
return $trilearnAction$
```

Algorithm 5.2: Pseudo code for the functions mapping our classes to the Trilearn Team's implementation of the CG algorithm.

Chapter 6

Experiments and Results

In this chapter we describe the experiments we did to test the effects of our extensions to the UvA Trilearn Soccer Simulation Team. Section 6.1 describes a simple experiment to show the effect of the `extra_y` *parameter on the space available to the Trilearn Team. Section 6.2 describes the various experiments we did related to coordination. In Section 6.2.1, we describe the experiment we did to determine the effect of changing the utilities of our value rules on our agents' behavior. In Section 6.2.2, we describe the experiments we did to determine the effects of coordination on our players' passing behavior. Finally, in Section 6.3, we look at the execution time of the CG algorithm and determine which factors influence this time. We further show whether the CG algorithm is fast enough to be used in the RoboCup competition.*

6.1 Space

Our first experiment aims to show the effects of the `extra_y` parameter in the UvA Trilearn Team's formation file. As mentioned in our discussion of the `extra_y` parameter in Section 5.1, it was introduced to create 'width and depth' in the team's use of the soccer field, thus increasing the space available to the players. In this experiment we compare the avarage space available to the UvA Team during several soccer matches. In half of the matches the `extra_y` parameter is used, in the other half it is set to zero and thus is ignored.

Recall the definition of space from Section 3.3.2:

> "...space on a soccer field is the gaps between and around opponents, the areas on the field that hold no players of the opposing team."

Using this definition, we calculate the space available to the UvA Team as follows: First we draw an imaginary rectangle around our players. The extremes

of this rectangle are the left-most player, the right-most player, the forward nearest to the opponent goal and the goalkeeper. The area around the rectangle is also space, but since there are no players in it, it is useless to our team and thus is not taken into account when calculating the space available to our team [Catlin, 1990].
We then draw circles around the opponent players within the rectangle. The radius of these circles we defined to be the distance a homogenous player may cover in the space of one cycle. We subtract the area of each of these circles (taking into account possible overlapping circles and circles not completely inside the rectangle) from the area of the rectangle. The resulting number is the space available to our players (for illustration, see Figure 6.1).

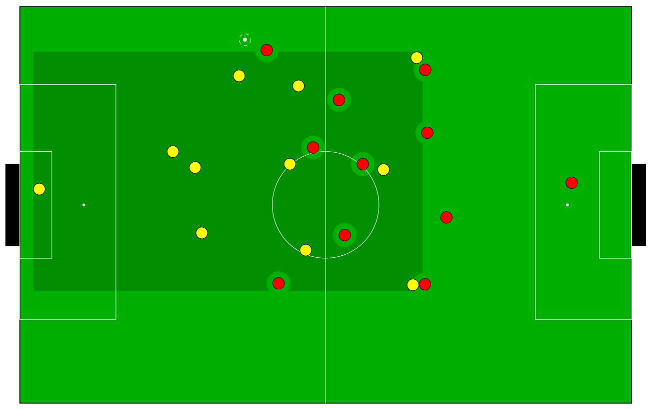


Figure 6.1: Example of how space was calculated during Experiment 1. The space available to our team is indicated in a darker green color.

In our experiment, we first set up the UvA Trilearn team to play three matches each against three different opponents, without using the `extra_y` parameter (i.e. with it set to zero), and calculated the average space available to our team, as described above. We then repeated this while using the `extra_y` parameter. The results are shown in Table 6.1.

The results in Table 6.1 show that the addition of the `extra_y` parameter has the desired effect: the average space available to our team increases. However, this result is only part of what we tried to achieve. By spreading out our players, we wanted to force the opposing team to do the same, resulting in larger gaps between their players, and their defense in particular. This effect cannot be surmised from the results in Table 6.1. While the average number of goals scored in a match did not decrease, neither did it increase due to a possibly weakened defense.

There are several explanations to why using the `extra_y` parameter did not have the desired effect. Firstly, the opponent teams we played against during the experiment did not respond to our players spreading out as we had wished. Therefore their defense remained the same as in the situation where we did not

[1]The average space taken over all matches (i.e. the average space calculated over 54.000 cycles).

	not using `extra_y`		using `extra_y`	
team name	**score**	**avg. space %**	**score**	**avg. space %**
FC Portugal	10 - 0	37.04 (± 6.85)	10 - 0	39.35 (± 8.33)
	10 - 0	36.39 (± 7.23)	8 - 0	38.70 (± 7.73)
	6 - 0	39.45 (± 6.40)	6 - 0	39.85 (± 8.25)
YowAI	11 - 0	36.90 (± 5.35)	14 - 0	39.43 (± 4.61)
	7 - 0	36.78 (± 5.38)	11 - 0	38.40 (± 5.26)
	12 - 0	37.49 (± 5.49)	10 - 0	39.43 (± 4.61)
AT Humboldt	23 - 0	34.86 (± 8.48)	21 - 0	35.71 (± 8.96)
	24 - 0	34.85 (± 8.27)	18 - 0	36.85 (± 8.44)
	23 - 0	33.68 (± 8.06)	22 - 0	36.08 (± 8.75)
avg. space %[1]		36.38 (± 6.83)		38.20 (± 7.21)

Table 6.1: Results for Experiment 1. Note that each match lasts 6000 cycles, and the average space is calculated over each cycle.

use the `extra_y` parameter. While our attackers were, in most cases, better able to set up attacks from the wings, they were not more effective compared to the original configuration.

Another explanation is that to make the most of the effect of the `extra_y` parameter, its values should be adapted to the playing style and formation of each individual opponent. This requires either an extensive knowledge of those opponents or the careful analysis of their playing style in earlier matches, or both. Due to time constraints we were unable to use either approach, thus limiting the effect of the `extra_y` parameter by using a fixed configuration throughout the experiment.

Finally, the changes to the `extra_y` parameter were kept deliberately small. Our belief was that changing the `extra_y` parameter too greatly would result in a loss in performance. This is because the distances between our players would become too great, and thus the chance for a successful pass would become smaller.

6.2 Coordination

In our second experiment we aimed to show the effects of using the CG algorithm and the rules we defined in Section 5.2 within the UvA Trilearn team. For this experiment we selected a subset of the rules from Section 5.2.3. We converted these rules to XML as described in Section 5.3.1.

In our rulebase, several rules have a value of -1. The values of these rules, which involve passing and player movement, are computed in real time. The other rules have constant values. Note that the rules involving the *interceptor* and *passive* roles (rules 14 and 15) have a value of 1.0, to indicate that these rules should always be executed by the players with the appropiate roles. This is quite natural, since we do not want an *interceptor* to do anything else but intercept the ball, and we do not want a *passive* player to do anything but move to its strategic position. Likewise, rule 12, in which a player in front of the goal tries to score, also has a value of 1.0. Finally, rule 11 has a value of 0.95. Because of the nature of the values computed in real time, this means that an agent that has the role *passer* will choose to dribble (if the context allows the rule) if he cannot pass in the directions *north*, *north-east*, or *north-west* of one of his receivers. This causes an emphasis on dribbling in the behavior of our players.

After instantiation, the 7 rules in our XML rulebase amount to 274 rules in total.

6.2.1 Experiment 1: Utilities

In the first part of the experiment we wanted to show the effect of changing the utilities of our value rules on the overall behaviour of our players. Using the rulebase and utilities as described above, we set up the UvA Trilearn Team to play against a team called *De Meer 5*. This team is one of the earlier versions of the UvA Trilearn Team and is described in Section 9.5.1 of [de Boer and Kok, 2002]. *De Meer 5* is not a strong team with respect to the teams available at the time of writing this thesis (the difference, for example, with the UvA Trilearn Team that won the World Championship in Padova is quite stunning), but in the light of our expectations and aims in this experiment, the team's performance was sufficient.

We played our team against *De Meer 5* for five full matches, and calculated how many times each action (see Table 5.2 for the complete list of actions) was selected by each player type. The results of our calculations is shown in Table 6.2.

It is clear to see that most of the action took place in the offensive zone of the field. The defenders (player types 2, 3, and 4) did little more than move to their strategic position, which indicates that they were assigned the role *passive* for most of the match. Looking at the midfielders and forwards, a greater emphasis is on passing and intercepting. Especially noteworthy is the high percentage of dribbling by the wing attackers. A favorite strategy of our team was to pass the ball to the wings, who would dribble past the defenders (and, unfortunately, often past the end-line, as well, due to the fact that only dribbling in the direction *north* was defined) and pass the ball back to the central attacker.

In the second part of the experiment we changed the utility of rule 11, the rule that defines the dribbling behaviour of our players, to 0.5. The resulting action percentages, calculated again over five full matches against *De Meer 5*, are shown in Table 6.3. As expected the overall amount of dribbling decreases, while there is an increase in the pass-related actions `move-to`, `pass-to`, and `intercept`. Also, looking at the percentages for the `move-to-strategic-position` action, the activity on the field is spread more due to the fact that passing backwards now yields an higher utility than dribbling forwards.

These results show that changing the utilities of the various rules results in

	action %					
pl. type	`move-to`	`pass-to`	`dribble-to`	`intercept`	`score`	`strat.-pos.`
2	3.48	0.58	0.28	2.59	0.00	93.07
3	4.81	0.70	0.74	3.35	0.00	90.40
4	6.53	0.78	0.97	2.70	0.00	89.02
5	27.86	2.07	1.47	5.71	0.01	62.88
6	17.39	1.41	1.41	5.11	0.02	74.65
7	24.88	3.27	6.78	9.96	0.12	55.00
8	26.33	3.41	2.06	7.82	0.01	60.36

Table 6.2: The first set of results from Experiment 1.

	action %					
pl. type	`move-to`	`pass-to`	`dribble-to`	`intercept`	`score`	`strat.-pos.`
2	6.45	0.73	0.00	2.94	0.00	89.88
3	3.91	0.52	0.22	2.18	0.00	93.17
4	12.16	1.96	3.00	5.96	0.00	76.93
5	19.57	2.06	0.00	4.99	0.00	73.39
6	21.85	1.97	0.92	6.46	0.00	68.81
7	20.09	3.27	2.27	8.35	0.00	66.02
8	31.59	2.90	1.34	5.91	0.00	58.27

Table 6.3: The second set of results from Experiment 1.

different behavior of our players on the field. Since the changes were made solely in the XML document describing our rules, this also shows that our aim of *flexibility* has been realized, at least in the aspect of changing the behavior of our soccer player by changing the utilities of our rules.

6.2.2 Experiment 2: Passing

In our second experiment we observed the passing behaviour of our agents. We used the same rulebase as used in the first part of Experiment 1.

We played ten games against ourselves, with one team using coordination and the other without using coordination in passing. The former team used the full rulebase, while the latter used the rulebase with rule 1 edited to remove the receiving player's `move-to` action. In this way, the coordination between passer and receiver was removed, disabling the ability for the receiving player to 'think ahead' on the action of the passer as discussed in Section 5.2.3 while describing the passing rules. For the same reason, rules 2 and 3 were removed. After instantiation, the rulebase for the non-coordinating team consisted of 94 rules.

Table 6.4 clearly shows the effect of coordination on the passing behavior of our agents. The successful passing percentage for the team using coordination

[2]The percentage the ball was on the offensive half of the field.

	using coordination	no coordination
wins	8	0
draws	2	2
losses	0	8
avg. score	1.80 (± 1.40)	0.00 (± 0.00)
possession %	53.78 (± 3.95)	46.22 (± 3.95)
passing %	88.53 (± 1.90)	60.62 (± 4.56)
ball pos. %[2]	66.86 (± 5.58)	33.14 (± 5.58)

Table 6.4: Results of ten games against ourselves, with and without coordination in passing.

was 88.53%, where the percentage of successful passes for team that did not use coordination was 60.62%. The number of passes for each team did not differ much. Another apparent effect of using coordination in passing is shown by the ball position percentage: two thirds of the match the ball was in the offensive half of the team using coordination, which resulted in more scoring opportunities (and consequently more goals) for that team.

Also, to see how our team would hold against different opponents, we played our team for five full matches each against *de Meer 5* and the UvA Trilearn Team that was used during the World Championships in Fukuoka in 2002. We generated the successful passing percentages and ball possession percentages using *ProxyMike*, a program that analyzes games played through the logplayer, generating a wide range of statistics.

	using coordination		no coordination	
	de Meer 5	UvA Fukuoka	de Meer 5	UvA Fukuoka
wins	4	5	2	5
draws	1	0	3	0
losses	0	0	0	0
avg. score	2.00 (± 1.41)	5.57 (± 3.15)	0.40 (± 0.49)	5.80 (± 1.60)
possession %	70.14 (± 1.47)	57.27 (± 4.15)	66.40 (± 1.39)	54.93 (± 1.60)
passing %	89.45 (± 0.73)	93.11 (± 1.24)	71.39 (± 2.78)	85.37 (± 2.47)
ball pos. %	84.23 (± 1.83)	75.89 (± 7.38)	49.51 (± 7.29)	74.45 (± 3.93)

Table 6.5: Results of five games each against two different opponents, with and without coordination in passing.

The first thing to note in Table 6.5 is that against *de Meer 5*, although the ball possession percentage was much higher than that against the Fukuoka team, our team was not able to score as much. This is due to a difference in playing style in the defense between *de Meer 5* and the Fukuoka team. *De Meer 5* plays much more aggressively, hunting the ball as opposed to waiting for it. The defense of the Fukuoka team is more stationary, waiting for the ball to get close while positioning itself strategically between the ball and the goal, and the forwards and the goal. In this situation the advantage of coordination becomes clear: by knowing where the pass will arrive before it is made, our players were able to cut through the defense of *UvA Fukuoka* quite easily, resulting in a large number of goals. Our team was less successful against the defense of *de Meer 5*, who left us with less room to move in and less time to move into a strategic position. When we compare the scoring results against *UvA Fukuoka* using coordination and without using coordination, there is practically no difference. This is due to the fact that dribbling in the *UvA Trilearn* team is very well developed. Thus, if a player would pass the ball in front of the defense, the receiving player could make use of the space created by one of the defenders moving to block the passer to dribble past the defense. This tactic clearly does not work against the aggressive defense of *de Meer 5*.

Looking at the successful passing percentages, we notice a greater drop in performance when using no coordination against *de Meer 5* than against *UvA Fukuoka*. This again can be ascribed to the aggressiveness of *de Meer 5*. When the passing performance is lower, an aggressive team will be able to intercept the ball more often compared to a less aggressive team. This also resulted in *de Meer 5* being able to pressure our team more when we did not use coordination, resulting in the change in ball position percentage and the greater amount of drawn games.

From the experiments described above, we first conclude that using coordination in passing results in a better performance of our team. Our team makes fewer mistakes in passing, and thus is able to pressurize the opponent team, resulting in more wins. What we also have seen is that using coordination in passing is relatively more successful against teams that deploy a stationary defense that is less aggressive in hunting the ball. While the increased ball possession is no surprise in this situation, the ability to cut through the defense by 'thinking ahead' on the passes gives our team a distinct advantage and more chances of scoring against these kind of teams.

6.3 Execution Time

Throughout the experiments described in Section 6.2, we have measured the execution times of the different processes of the CG algorithm. We wanted to determine which factors influenced the total execution time of the CG algorithm, and whether the algorithm would be fast enough to be usable in the various RoboCup competitions.

Table 6.6 shows the execution times for the various processes of the CG algorithm with a rulebase of 274 rules for each agent. This was the case when our team used coordination in passing. Even with 274 rules, the average execution time of the CG algorithm is 5.7230 milliseconds. When we recall that according to the experiments performed in Chapter 5 of [de Boer and Kok, 2002] the CG algorithm has 8 milliseconds to complete, we can conclude that, using this rulebase, our CG algorithm would be fast enough to be used in the RoboCup competitions.

To determine which factors contribute most to the execution time of the CG

[3]The games were played on a Pentium III 1000Mhz machine with 256MB of RAM.

process	**avg. execution time (ms)**[3]
assigning roles	0.0471 (± 0.0004)
context generation	1.1573 (± 0.0393)
updating context	1.8603 (± 0.0650)
updating values	0.0515 (± 0.0030)
elimination	2.6069 (± 0.3442)
total	5.7230 (± 0.4369)

Table 6.6: Average execution times in milliseconds for the different parts of the CG algorithm during Experiment 2, calculated over 20 matches. The total number of rules for each agent was 274.

---

algorithm, we compare the results in Table 6.6 with those shown in Table 6.7. In this table, the execution times are calculated using a rulebase of 94 rules for each agent, the rulebase used by our team during the part of Experiment 2 where our team used no coordination in passing.

process	**avg. execution time (ms)**
assigning roles	0.0462 (± 0.0004)
context generation	0.9627 (± 0.1468)
updating context	0.3467 (± 0.0233)
updating values	0.0333 (± 0.0017)
elimination	0.4267 (± 0.0157)
total	1.8151 (± 0.1591)

Table 6.7: Average execution times in milliseconds for the different parts of the CG algorithm during Experiment 2, calculated over 20 matches. The total number of rules for each agent was 94.

Comparing Table 6.6 and Table 6.7, we note that there is practically no difference in execution time for the assignment of roles to our agents. This makes sense, since for every situation the roles are distributed the same: one agent gets either the *passer* or the *interceptor* role, two agents are assigned the *receiver* role, and the rest of the agents are *passive*. For the same reason there is practically no difference in execution time for the generation of contexts. In each match, more or less the same contexts apply, in more or less the same distribution.

Updating the context is the first process that changes a lot when using a smaller rulebase. This is because each rule in the rulebase must be checked against the generated context to determine whether it applies or not. The grouping of rules with respect to the agents' roles, as we described in Section 5.2.3, has great effect on the execution time of this process. Without grouping, each agent has to update all rules for every agent. Using grouping, each agents only updates the rules that belong to his role. Using grouping, the execution time for this process is rougly 30 times faster.

Updating the values of the rules that survive updating the context is trivial again, taking very little time. Theoretically, the execution time depends on the amount of rules in our XML document that require that their values are computed during each cycle, but the difference is very small.

Lastly, the actual elimination algorithm takes the most time, as expected. The execution time for the elimination of rules depends largely on the amount of rules that survive updating the context, and the amount of rules therein that involve other agents (e.g. rule 16, the only rule for the *passive* role, involves no other agents and thus will not add much to the elimination time). As illustration, Table 6.8 shows the number of applicable rules that survived updating the context with the respective execution time for the elimination algorithm. Note that the values in the bottom row are calculated where the team used no coordination, and thus there are no rules involving other agents.

avg. applicable rules	**avg. elimination time (ms)**
25.00	4.5251
18.70 (± 0.46)	2.6069 (± 0.3442)
10.75 (± 0.54)	0.4267 (± 0.0157)

Table 6.8: The average number of applicable values rules and the respective execution time of the elimination algorithm. The values in the top row represent a worst case scenario for the rulebase we used in our experiments.

In this table, the values in the top row were calculated by playing our team, using coordination, against no opponent. In this way, most contexts were true, and thus most rules survived updating the context, creating a worst case scenario in terms of execution time. As we assumed that there would be practically no variation in values over the course of several games, we calculated these values over 750 cycles, which means that the CG algorithm was executed roughly 7500 times. As we calculated the standard deviation using several games, we weren't able to compute it for these values. It can be assumed, though, that the deviation is very small.

For completeness, the complete table depicting the execution times in a worst case scenario is given in Table 6.9.

process	**avg. execution time (ms)**
assigning roles	0.0469
context generation	0.7822
updating context	2.5653
updating values	0.1212
elimination	4.5251
total	8.0406

Table 6.9: Execution times for the different processes of the CG algorithm in a worst case scenario. The total number of rules for each agent was 274.

6.3.1 Adding more rules

To see what effect a larger rulebase would have on the execution time of the CG algorithm, we created a new rulebase by copying rule 1 and 3 eight times each. Thus our XML document contained 16 rules, eight of which involved two agents. After instantiation, the total number of rules for each agent was 1710. The average number of applicable rules after updating the context was 24.

process	**avg. execution time (ms)**
assigning roles	0.0288
context generation	0.5935
updating context	6.1769
updating values	0.1150
elimination	32.7752
total	39.6895

Table 6.10: Execution times for the different processes of the CG algorithm for a larger rulebase. The total number of rules for each agent was 1710.

Understandably, the execution time for updating the context increases with the larger number of rules for each agent. However, this increase is not very large (a factor 2.4) with respect to the increase in the number of rules from 274 to 1710 (a factor 6.2).

Compared to the execution times in Table 6.9, the execution time for the actual elimination process is much higher, while the average number of applicable rules after updating the context is largely the same. This shows that, while the execution time of elimination process does depend on the number of applicable rules left, it depends also on the number of rules that involve 2 or more agents. Where the rulebase used in Table 6.9 contained 1 such rule (rule 1), the rulebase used in Table 6.10 contained 8 such rules (8 copies of rule 1). The execution time is roughly 5 times higher.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

Soccer is a complex game, and though its rules are well-defined, it is very hard to create a mapping from real soccer situations to robotic soccer situations. However, certain fundamental rules seem common in the decision process of real soccer players. In this thesis, we have researched real soccer strategies and situations to find these rules, and have used Coordination Graphs to implement these rules into the UvA Trilearn Robotic Soccer Simulation Team.

Our main objectives were to convert and implement the real soccer rules we found in such a way that both flexibility with respect to adding to and changing rules and speed with respect to the execution time of the CG algorithm were realized. From our experiments, we may conclude that to a certain extent we have succeeded in both goals.

Coordination Graphs are a relatively simple and effective way to represent and solve the coordination problem in multiagent systems. The research presented in this thesis shows that CGs can be used in real-time, highly dynamic environments. We have determined that the number of rules for each agent within the CG does not influence the speed as much as the number of agents involved in each rule, which advocates for keeping the rules as simple as possible. Our research shows that even with relatively simple rules, complex behavior can be defined.

7.2 Future Work

Future research based on this thesis can be divided into two sections:

First, research may be done to further implement the CG algorithm and the rules we presented in Section 5.2 into the UvA Trilearn team in particular and robotic soccer teams in general. Due to a lack of time the experiments in this thesis did not address using the coordination rules in the defense of the Trilearn team, and we are curious to see the results of research in this area. Further, we believe that some of our rules might be refined more, or new rules created, so that a more controlled behavior might be specified.

Second, there are some interesting possibilities in the area of reinforcement learning that might be applied to CGs in general. Learning the utilities of the value rules is one example. Another, more advanced example is the use of *hybrid systems*[1] to actually learn the rules used in the CG algorithm. Hybrid systems combine symbolic (rule-based) and connectionist (neural network-based) approaches to solving problems, combining the strengths of both worlds. Also, using CGs in other fields is an open field of research. Examples would include traffic control, robot exploration, and many more.

[1]http://www.comp.nus.edu.sg/~pris/HybridSystems/HybridSystemsIndex.html

Bibliography

[Catlin, 1990] Catlin, M. G. (1990). *The Art of Soccer*. Soccer Books, St. Paul, MN USA.

[de Boer et al., 2002] de Boer, R., Kok, J., and Groen, F. (2002). UvA Trilearn 2001 Team Description. In *Robocup-2001: Robot Soccer World Cup V*. Springer Verlag, Berlin.

[de Boer and Kok, 2002] de Boer, R. and Kok, J. R. (2002). The Incremental Development of a Synthetic Multi-Agent System: The UvA Trilearn 2001 Robotic Soccer Simulation Team. Master's thesis, University of Amsterdam, The Netherlands.

[Fagin et al., 1995] Fagin, R., Halpern, J. Y., Moses, Y., and Vardi, M. Y. (1995). *R*easoning about Knowledge. The MIT Press, Cambridge, MA.

[Foroughi et al., 2001] Foroughi, E., Heintz, F., Kapetanakis, S., Kostiadis, K., Kummeneje, J., Noda, I., Obst, O., Riley, P., and Steffens, T. (2001). RoboCup Soccer Server User Manual: for Soccer Server version 7.06 and later. At http://sourceforge.net/projects/sserver.

[Glanville, 1979] Glanville, B. (1979). *A book of Soccer*. Oxford University Press, Oxford, England.

[Guestrin et al., 2002a] Guestrin, C., Koller, D., and Parr, R. (2002a). Multiagent planning with factored MDPs. In *A*dvances in Neural Information Processing Systems 14. The MIT Press.

[Guestrin et al., 2002b] Guestrin, C., Venkataraman, S., and Koller, D. (2002b). Context specific multiagent coordination and planning with factored MDPs. In *P*roceedings of the Eighteenth National Conference on Artificial Intelligence, Edmonton, Canada.

[Kitano and Asada, 1998] Kitano, H. and Asada, M. (1998). 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)*.

[Kitano et al., 1997] Kitano, H., Asada, M., Kuniyoshi, Y., Noda, I., and Osawa, E. (1997). RoboCup: The Robot World Cup Initiative. In *Proceedings of the First International Conference on Autonomous Agents (Agent-97)*.

[Kok et al., 2002a] Kok, J. R., de Boer, R., Vlassis, N., and Groen, F. (2002a). UvA Trilearn 2002 Team Description. In Kaminka, G., Lima, P., and Rojas, R., editors, *R*oboCup 2002: Robot Soccer World Cup VI, page 549, Fukuoka, Japan. Springer-Verlag.

[Kok et al., 2002b] Kok, J. R., Spaan, M. T. J., and Vlassis, N. (2002b). An approach to noncommunicative multiagent coordination in continuous domains. In Wiering, M., editor, *B*enelearn 2002: Proceedings of the Twelfth Belgian-Dutch Conference on Machine Learning, pages 46–52, Utrecht, The Netherlands.

[Kok et al., 2003] Kok, J. R., Spaan, M. T. J., and Vlassis, N. (2003). Multi-robot decision making using coordination graphs. In *P*roc. International Conference Advanced Robotics. To appear.

[Kok and Vlassis, 2003] Kok, J. R. and Vlassis, N. (2003). The pursuit domain package. Technical Report IAS-UVA-03-03, Informatics Institute, University of Amsterdam, The Netherlands.

[Lubbers and Spaans, 1998] Lubbers, J. and Spaans, R. R. (1998). The Priority/Confidence Model as a Framework for Soccer Agents. In *Proceedings of the Second RoboCup Workshop*, pages 99–108, Paris.

[Morrison, 2001] Morrison, M. (2001). *HTML and XML for beginners*. Microsoft Press.

[Reis and Lau, 2001] Reis, L. P. and Lau, J. N. (2001). FC Portugal Team Description: RoboCup-2000 Simulation League Champion. In Stone, P., Balch, T., and Kraetszchmar, G., editors, *RoboCup-2000: Robot Soccer World Cup IV*, pages 29–40. Springer Verlag, Berlin.

[Riley et al., 2000] Riley, P., Stone, P., and Veloso, M. (2000). Layered Disclosure: Revealing Agents' Internals. In *Proceedings of the Seventh International Workshop on Agent Theories, Architectures and Languages (ATAL-2000).*

[Rumbaugh et al., 1999] Rumbaugh, J., Jacobson, I., and Booch, G. (1999). *The Unified Modeling Language Reference Manual*. Addison Wesley.

[Russell and Norvig, 1995] Russell, S. J. and Norvig, P. (1995). *Artificial Intelligence: A Modern Approach.* Prentice Hall, Englewood Cliffs, NJ.

[Spaan et al., 2002] Spaan, M. T. J., Vlassis, N., and Groen, F. C. A. (2002). High level coordination of agents based on multiagent Markov decision processes with roles. In Saffiotti, A., editor, *I*ROS'02 Workshop on Cooperative Robotics, Lausanne, Switzerland.

[Vlassis, 2003] Vlassis, N. (2003). A concise introduction to multiagent systems and distributed AI. Informatics Institute, University of Amsterdam. http://www.science.uva.nl/˜vlassis/cimasdai.