Learning to Co-operate in Multi-Agent Systems

K. Kostiadis

A thesis submitted for the degree of Doctor of Philosophy

Department of Computer Science

University of Essex

15th March 2002

Keywords: Artificial Intelligence, Machine Learning, Multi-Agent Systems, Reinforcement Learning, Multi-Agent Co-operation, Emergent Co-operation, Agent Architectures, Real-time Agents, Multi-threaded Agents, Design Patterns, Sparse Distributed Memories, Robotic Soccer.

Στην οικογένειά μου.

Abstract

In recent years, two major areas of computer science have started converging. Artificial intelligence research is moving towards realistic domains requiring real-time responses, and real-time systems are moving towards more complex applications requiring intelligent behaviour. To that extend, this thesis contributes techniques for building agents in a particularly complex class of real-world domains. More specifically, this thesis addresses the question of whether agents can learn to become individually skilled and also learn to co-operate in the presence of both teammates and adversaries in a complex, real-time, noisy environment with no communication.

The first step towards this investigation is the analysis of the functional requirements specification for agents wishing to operate in such environments. Following this, an agent architecture is proposed, which enables agents to combine reactive real-time responses together with deliberative intelligent decisions. A multi-threaded framework is also presented, which allows agents to operate with bounded response times.

Once the core components within the agent have been described, the focus turns to the agent's behaviour. Starting from its perception, this thesis demonstrates how a sparse distributed memory model can be used as a generalisation component for tasks that involve large state spaces. In addition, a simple task is used to demonstrate how reinforcement learning can lead to co-operative problem solving via intelligent action selection.

Finally, this thesis demonstrates how the proposed generalisation and action selection methods can be combined in order to produce a decision-making module suitable for noisy, real-time, collaborative and adversarial domains that involve large state spaces. A complex learning task is used to demonstrate that agents can learn to co-operate via their action selection mechanisms even when they have no knowledge of the concept of co-operation. Experimental results demonstrate how the learned policies outperform fixed, hand-coded ones.

Acknowledgements

I would like to thank all the people who have helped me, both directly and indirectly, during my three years as a graduate student here at Essex University. Without their support and guidance this work would not have been possible.

First and foremost, I would like to thank my advisor Huosheng Hu. Without his enthusiasm, encouragement, support, and guidance, this thesis would not have happened. Special thanks also go to the members of my supervisory board, Jim Doran and Libor Spacek for their useful comments during the last three years. I would also like to thank Paul Scott for his feedback on my initial learning experiments.

Apart from the academic staff, I would also like to thank the computer officers and technical staff here at Essex. In particular, Mike Millman, Graham Clarke, Ian Boddington, and Simon Moore have all been very helpful during the last three years. Also, thanks go to our administrative staff, Marisa Bostock, Jennifer Dixon, Chris Jennings, and Julie Peirson.

Very special thanks go to the fellow Wizards in the lab. Matthew Hunter, Nikos Kalyviotis, and Mike Seabrook have all done great work for the competition versions of the Essex Wizards team.

During my graduate years I was very lucky to be part of a great research community. All the people within the RoboCup domain have been great and it would require numerous pages to put all their names down. I would like to thank them all and in particular the people who volunteered to develop and maintain the soccer server, the people who volunteered to write the soccer server user manual, and all the people who took the time to answer questions in the mailing list. A few people I would like to thank in particular are Tom Howard (who also proof-read this thesis), Itsuki Noda, Patrick Riley, and Peter Stone.

I would also like to thank my MSc dissertation advisors Kerstin Dautenhahn and Kevin Warwick, as they were the ones that introduced me to the RoboCup domain during my graduate year in Reading University.

In addition, I would like to thank Leemon Baird, Pentti Kanerva, and Richard Sutton. Although I have never had the pleasure of meeting them in person, their comments and suggestions have been very useful during the development of the Essex Wizards teams.

Finally I would like to thank the Department of Computer Science, University of Essex, for the financial support to this project.

On a more personal note, I would like to thank my fiancée, Amanda Berry, who stood by me during all my student years, sharing both my enthusiasm and the brunt of my frustrations. My parents and my sister have always been, and continue to be there for me at all times.

Contents

1 Introduction 1

1.1 Overview 1

1.2 Objectives and Approach 2

1.3 Motivation 3

1.4 Contributions 5

1.5 A Guide to the Thesis 6

2 The RoboCup Initiative and the Soccer Server 9

2.1 Overview 9

2.2 The RoboCup Synthetic Agent Challenge 10

2.3 The RoboCup Soccer Server 13

2.3.1 Overview of the Server 14

2.3.2 The Simulated World 18

2.3.3 Client/Server Interface 22

2.3.3.1 Agent Perception 23

2.3.3.2 Agent Action 33

2.3.4 Sample Client/Server Interaction 39

2.4 Summary 42

3 Real-Time Multi-Threaded Agents 47

3.1 Overview 47

3.2 Agent Architecture 48

3.2.1 Requirements 48

3.2.2 Unix I/O Models 53

3.2.3 The Essex Wizards Architecture 54

3.3 Multi-Threaded Agents 60

3.3.1 Single-Threaded Agents 61

3.3.2 A Multi-Threaded Approach 66

3.3.3 Experimental Results 69
3.3.4 Single versus Multi Threaded Implementations 70
3.4 Design Patterns 71
3.4.1 Why Design Patterns 72
3.4.2 Designing a Multi-Agent System 74
3.4.3 Multi-Threaded Mediator Pattern 75
3.5 Discussion 78
3.6 Related Work 79

4 Kanerva Based Generalisation (KaBaGe) 83
4.1 Overview 83
4.2 Why Kanerva-Coding 85
4.3 A Brief Introduction to Kanerva's Memory Model 86
4.3.1 Neurons as Address Decoders 86
4.3.2 Sparse Memory 88
4.3.3 Distributed Storage 90
4.3.4 Storage and Retrieval of Sequences: Prediction 93
4.3.5 Memory for Patterns and Pattern Sequences 94
4.4 Implementing the Generalisation Module 95
4.4.1 Design and Implementation of KaBaGe 96
4.4.2 Prototype Construction 98
4.5 Experimental Results 100
4.6 Related Work 104

5 Learning to Act Using Reinforcement Learning 109
5.1 Overview 109
5.2 Why Reinforcement Learning 110
5.3 A Brief Introduction to Reinforcement Learning 111
5.3.1 Models of Optimal Behaviour 113
5.3.2 Difficulties in RL 114
5.3.3 RL Methods 118
5.4 Implementing a Q-learning Based Decision Making Module 120
5.5 The 2 Attackers versus 1 Defender Challenge 122
5.6 Experimental Results 124

5.7 Discussion 126
5.8 Related Work 129

6 Learning to Co-operate Using KaBaGe-RL 135
6.1 Overview 135
6.2 Why KaBaGe-RL 136
6.3 The 3 Attackers versus 2 Defenders Possession Challenge 137
6.4 Implementing KaBaGe-RL 141
6.4.1 KaBaGe and Feature Representation 142
6.4.2 Q-learning and Residual Algorithms 145
6.5 Experimental Results 149
6.6 Related Work 151

7 Conclusions and Future Directions 160
7.1 Overview 160
7.2 Contributions 162
7.3 Future Directions 163
7.3.1 Future Directions within RoboCup 164
7.3.2 Future Directions within the Contributed Algorithms 165
7.4 Concluding Remarks 167

Appendix A – List of Acronyms 168
Appendix B – Source Code 170
Appendix C – Server Parameters 171
Appendix D – References 173

List of Figures

1.1 Relevance of each chapter to the thesis contributions. 8
2.1 Overview of the server 15
2.2 The soccer monitor 16
2.3 The locations of the server's 55 visible markers 20
2.4 Client/Server interface 23
2.5 The view_angle parameter and its effect on visual sensations 31
2.6 Screen shots representing the initial and final stages of the sample interaction 43
3.1 Agent's main loop 52
3.2 The Essex Wizards architecture 55
3.3 Sensors I/O 56
3.4 Behaviours I/O 57
3.5 Actuators I/O 58
3.6 World-Model I/O 59
3.7 Parameters I/O 60
3.8 Example of a single-threaded agent 62
3.9 Pseudo-code implementation of I/O multiplexing 63
3.10 Pseudo-code implementation of signal driven I/O 65
3.11 Example of a multi-threaded agent 66
3.12 Pseudo-code implementation of a multi-threaded approach 68
3.13 Mediator Pattern 76
3.14 World-Model mutex operation 77
4.1 Address of a neuron 87
4.2 Sparse memory 89
4.3 Distributed storage 92
4.4 The KaBaGe module 96
4.5 Generalisation example 97
4.6 General 3-layer architecture using KaBaGe 101
4.7 Learning curves on first task 102

4.8 Performance versus representation size on the first task 103
5.1 Standard RL model 112
5.2 The agent's learning process 121
5.3 State graph of the 2v1 challenge 123
5.4 Initial states for the 2v1 challenge 124
5.5 Achieving a global goal by distribution of responsibilities 128
6.1 Screen shot from a 3v2 possession episode in a 20m x 20m region 138
6.2 The attacker's policy space 140
6.3 The state variables used by KaBaGe 143
6.4 KaBaGe links to all possible actions 144
6.5 Possession time versus hours of training 150

List of Tables

2.1 Characteristics of sensory information 23
2.2 The different actions available to agents 34
3.1 Cycle loss 69
5.1 Results for a random attacker 125
5.2 Results for a Q-learning attacker 125
C.1 Server parameters 171

Chapter 1

Introduction

1.1 Overview

Multi-agent systems (MAS) have recently emerged as an active subfield of artificial intelligence (AI). One key concept that differentiates MAS from other related disciplines is that of *co-operation*. Although there is no generally accepted definition for co-operation, for the purposes of this thesis co-operation is said to occur when *an agent's actions tend to achieve the groups' global goal.* This definition matches Doran's viewpoint on co-operation [Doran et al., 1997] and focuses on the actions of the agents and their goals, irrespective of how they arise. Also, like Franklin's definition of *emergent co-operation* [Doran et al., 1997], the agents are not required to intentionally co-operate for co-operative problem solving to occur.

Given the complexity of most MAS and the size of the state and action spaces, Machine Learning (ML) is an interesting area to merge with MAS. In the recent years a number of successful ML applications have been developed, ranging from master level synthetic backgammon players [Tesauro, 1994] to autonomous vehicles that can drive on public highways [Pomerleau, 1993]. Although it is not yet known how to make computers learn as well as people do, certain algorithms have been invented that are effective for certain types of learning tasks. For example in problems such as speech

recognition, algorithms based on ML outperform all other approaches that have been attempted until now. In complex multi-agent environments, ML has the potential to provide robust mechanisms for action selection that significantly improve the performance of an agent. In addition, ML based decision making can lead to efficient *co-operation* if this proves to be beneficial for the group.

This thesis contributes several techniques for developing a successful group of agents that is required to operate in a real-time, noisy, collaborative and adversarial environment towards the achievement of a global goal. As of yet, there has been little work in environments that have all of these characteristics.

1.2 Objectives and Approach

The principal question addressed in this thesis is:

Can agents *learn* to become individually skilled and also *learn to co-operate* in the presence of both teammates and adversaries in a complex, real-time, noisy environment with no communication?

More specifically, this thesis contributes an agent architecture together with a ML based decision-making module that improves the performance of a group of agents in domains with the following characteristics:

- A need for real-time decision making
- Sensor and actuator inaccuracy due to noise
- Limited sensing capabilities that result in large amounts of hidden state

- A group of agents with the same well defined global goal: teammates
- A group of agents with a conflicting well defined global goal: adversaries

Note that teammates may have the same global goal, but they may also have different local goals. Agents are assumed to have the following properties:

- Sensors that provide partial, noisy information regarding their environment
- The ability to process the sensory information and generate actions
- The ability to execute these actions (via inaccurate actuators), affecting the environment

The general approach to answering the principal question addressed in this thesis was to create an existence proof. In the remainder of this thesis, a multi-agent system is developed which exhibits co-operative problem solving via a ML based decision-making module in a real-time, noisy environment with the presence of both teammates and adversaries.

Nearly all the experiments presented in this thesis were carried out on the simulated version of the robotic soccer domain (also known as RoboCup). RoboCup serves as an ideal test bed for the contributions of this thesis as it provides an environment with all the desired properties.

1.3 Motivation

Perhaps the work most closely related to the one presented in this thesis is [Stone, 1998]. Although Stone addresses a similar research question, the approach to answering this

question is very different. [Stone, 1998] classifies RoboCup as a periodic team synchronisation domain and relies on communication between agents in order to execute multi-agent co-operative plans. These plans (also referred to as set-plays) are fixed, and known to all agents in advance. During normal operation, global triggers are used in order to determine which plan to invoke. Agents rely on communication in order to execute one of the pre-defined plans.

Inspired by their success, the author set out to construct a MAS consisting of agents that can *learn to co-operate* towards the achievement of a global goal. However, the purpose of this thesis is not to present a new co-operative framework. The study of evolution has demonstrated that even animals recognise the benefits of co-operation and, for example, hunt in groups. Therefore it should be possible for intelligent agents to demonstrate a form of *emergent co-operation* via their decision-making mechanisms (assuming they share a common global goal). Another difference with [Stone, 1998] is that none of the experiments in this thesis use explicit communication between the agents. This form of *non-communicating* MAS allows for the use of the proposed approach in domains where communication is very costly or even impossible. Also, systems where agents can communicate with each other have already been studied within the RoboCup domain [Stone, 1998], which is another reason this thesis focuses on non-communicating MAS.

On a more general level, nearly all researchers in the RoboCup domain have been inspired by the Dynamite test bed [Barman et al., 1993]. Their successful implementation of two non-holonomic robots pushing a ping-ping ball towards opposite goals on a walled field was the first real robotic soccer system [Stone, 1998]. Although Dynamite used a single robot on each team, it generated the idea of creating robotic soccer for studying real-time, collaborative and adversarial MAS. Moving from a single agent (per team)

system to a fully (11 versus 11) multi-agent one immediately introduced numerous research challenges.

1.4 Contributions

This thesis makes four distinct contributions to the fields of MAS and ML:

A generalised agent architecture suitable for complex real-time domains that require a deliberative decision-making process. The proposed model combines a standard *sense-think-act* cycle together with a *probabilistic world-model*. In addition, the architecture provides a *multi-threaded mediator*, which simplifies the inter-communication between the various components within the agent.

A multi-threaded implementation of the agent's *sense-think-act* cycle for improved performance and bounded response times. The proposed implementation considers three threads that operate concurrently (or in parallel if multi-processor hardware is available). The proposed model performs significantly better than the existing single-threaded ones, given the asynchronous sensing and acting rates of the agent and the slow communication links between the agent and the environment.

The KaBaGe-RL decision-making module, which combines a generalisation module together with reinforcement learning (RL) based action selection. The generalisation module is based on Kanerva's sparse distributed memory (SDM) and is responsible for expanding the original input representation received by the agent. The action selection

module is based on residual Q-learning and is responsible for both choosing an action and updating the weights of the function approximator.

The Essex Wizards Robotic Soccer System, which is a fully implemented and operational team of simulated robotic agents. The documented description of this research, together with the algorithmic details and the source code, should enable future researchers to build upon the successful implementation of the Essex Wizards team.

Although the last of these contributions is domain specific, the generalised descriptions of the first three together with their implementation details should allow for their use in other domains.

1.5 A Guide to the Thesis

This section describes the contents of each chapter together with guidelines as to which chapters are relevant to which contribution (figure 1.1).

Chapter 2 introduces the Robot World Cup Initiative and the Soccer Server. The description of the RoboCup Initiative also illustrates the research challenges that potential developers have to face. The description of the soccer server aims to give a better understanding of the interface between the agents and the environment, and also highlight some of the implementation difficulties that are normally encountered.

Chapter 3 describes the proposed agent architecture. Initially, the requirements of the agent are highlighted, given the timing constraints enforced by the environment.

Following this, the multi-threaded model for the execution of the *sense-think-act* cycle is presented. Experimental results show the superiority of such a model given the nature of the communication between the soccer server and the agents. Finally, the multi-threaded *mediator* pattern shows how the inter-module communication within the agent can be simplified.

Chapter 4 introduces the KaBaGe module. Initially, the need for generalisation in problems with large state and action spaces is discussed. Following this, a description of Kanerva's SDM is given together with the reasons as to why it was chosen as a function approximator. Finally, this chapter presents a detailed description together with implementation details of the KaBaGe module.

Chapter 5 describes the RL based decision-making module. The reasons for choosing RL over other ML methods are highlighted and a brief description of the challenges that arise from RL based decision-making are illustrated. Finally, the simple *2 Attackers versus 1 Defender* co-operative challenge is used to evaluate the performance of the action selection within the robotic soccer domain.

Chapter 6 introduces the KaBaGe-RL decision-making module. KaBaGe-RL combines the material presented in chapters 4 and 5 in order to produce a learning module suitable for complex real-time environments with large state spaces. The performance of KaBaGe-RL is evaluated using the *3 versus 2 Possession Football* challenge, a co-operative task involving 5 agents.

Chapter 7 summarises the contributions of this thesis and also outlines some of the promising directions for future work.

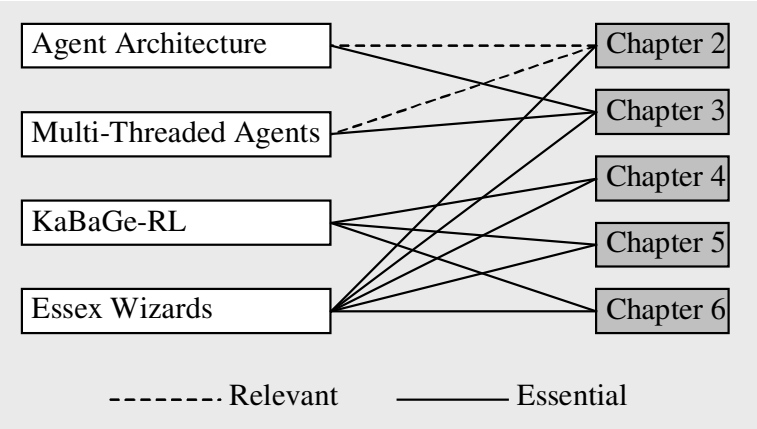


Figure 1.1: Relevance of each chapter to the thesis contributions.

Chapter 2

The RoboCup Initiative and the Soccer Server

2.1 Overview

Standard artificial intelligence (AI) problems have been the basic driving force for AI research [Kitano et al., 1997]. For example research on computer chess has lead to the discovery of powerful search algorithms. Other problems, such as the *Travelling Salesman* or the *8-Queens puzzle*, contributed to AI research by illustrating the essential difficulties involved in everyday reasoning. Criticisms against using such problems mostly focus on the simplicity of these tasks comparing to real-world problem solving. Unfortunately, solving real-world problems involves specific constraints, which are not necessarily common in other domains. In addition, research on usable real-world systems is often beyond the manpower and funding of many research groups [Kitano et al., 1995]. This makes it difficult to compare results of different techniques that have been applied in order to solve real-world tasks. The Robot World Cup Initiative is designed to tackle such difficulties. It is intended as a standard problem that is both realistic and at the same time affordable for many research groups. More specifically, RoboCup is an attempt to foster AI and intelligent robotics research by providing a standard problem where wide range of technologies can be integrated and examined [Kitano et al., 1997]. Some of the fields covered include machine learning, agent architectures, real-time planning and

reasoning, multi-agent co-operation, and sensor fusion. Although RoboCup involves a number of different events and competitions (e.g. RoboCup Rescue, RoboCup Junior, etc.) the focus of this thesis is on the robotic soccer competitions, and more specifically the *software robotic soccer* competition (also known as the *simulation league*).

This chapter introduces the challenges of the RoboCup domain together with the main ideas behind the Robotic World Cup Initiative [Kitano et al., 1995]. A description of the RoboCup soccer server (or just *server* for short) is also presented. More specifically, section 2.2 illustrates the main research challenges faced by synthetic RoboCup agents [Kitano et al., 1997], while section 2.3 highlights the key features of the RoboCup soccer server [Chen et al., 2001] [Noda et al., 1998] [Stone, 1998]. The aim of this chapter is to illustrate the challenges that need to be addressed in order to build a successful multi-agent system (MAS) for the RoboCup and similar domains. In addition, a better understanding of the environment and the interface between the server and the agents will ease the explanation of the experimental results presented in subsequent chapters of this thesis.

2.2 The RoboCup Synthetic Agent Challenge

The fundamental issue for researchers who wish to built a team for the RoboCup domain is to design a multi-agent system that behaves in real-time, performing reasonable goal-directed behaviours [Kitano et al., 1997]. Note that the size of the state space is prohibitively large for anyone to hand-code all possible situations and actions. Therefore, the use of machine learning techniques is essential in order to build a successful team. The main challenges a synthetic agent faces are:

- Individual learning
- Multi-agent co-operation
- Opponent modelling

This thesis addresses the first two challenges, i.e. individual learning and multi-agent co-operation. However, these challenges are not treated separately. The main idea is that at an abstract level, action selection of individual agents should lead to co-operation if this proves to be beneficial for the team. It has been shown that groups of agents working together can derive more efficient solutions in terms of energy, costs, time and quality [Kube and Zhang, 1992] [Noreils, 1993] [Altenburg, 1994] [Mataric, 1995]. The distribution of responsibilities and multiple roles in co-operative frameworks is a common feature. Each agent in a team has an individual role and therefore a set of responsibilities [Ch'ng and Padgham, 1997a] [Ch'ng and Padgham, 1997b]. The aim of this thesis is not to present a new co-operative framework. Instead, the purpose is to show that an intelligent agent working in a team should be able to *learn* that *co-operation* is beneficial. Experimental results presented in subsequent chapters demonstrate how co-operation *emerges* through an agent's decision making process, even when the agent has no knowledge of the concept of co-operation. A brief description of the three main challenges in the RoboCup domain is given below.

Individual Learning

Learning is an essential aspect of any intelligent system. In the RoboCup domain there are numerous opportunities for learning. For example an agent could be trained to intercept a moving ball, or learn whether to shoot or pass when in possession of the ball. The main classification for learning schemes is based on whether the agents are trained

off-line and remain fixed during a game, or learn on-line and adjust their policies depending on the opponents performance. Both types of learning have proven equally useful. For example [Stone, 1998] demonstrated how a decision tree can be trained off-line and then used for pass evaluation during a game. [Kostiadis and Hu, 1999] [Kostiadis and Hu, 2001] showed how on-line learning methods can be used for decision making. Although both types of learning are equally important, the focus of this thesis is on on-line learning. There are a number of reasons this kind of learning is preferred. Initially, it is potentially more robust since errors in the training set can be corrected during normal operation. In addition, on-line learning is necessary in order to learn and track time-varying functions and therefore continue to adapt to a changing environment. Note that this property is particularly important in the RoboCup domain since each game involves different opponents that use different formations and strategies. Finally, a more general property of on-line learning is that training data can often be generated easily and in great quantities when a system is in operation. As mentioned in [Sutton and Whitehead, 1993], on-line learning is essential in order to obtain *learning* systems as opposed to merely *learned* ones.

Multi-agent Co-operation

The challenge of multi-agent co-operation addresses issues of real-time planning and execution of co-ordinated actions involving multiple agents in a dynamic adversarial environment. Some of the typical issues that arise include dynamic changes in a team's goals, unexpected inabilities to fulfil responsibilities by team members, and unexpected discovery of opportunities. Teamwork in such a complex, dynamic multi-agent domain requires flexible co-ordination and communication in order to tackle these difficulties and to overcome the uncertainties of the environment. The generality of such co-operative

frameworks for non-RoboCup applications is also an important factor. Note that pre-planned, domain specific co-ordination will fail to provide the flexibility that is required. This is mainly due to the difficulty to anticipate and pre-plan for all possible co-ordination failures. Also note that planning on the fly is extremely difficult given the choice of actions and the size of the state space. Implementations of multi-agent co-operative plans in the RoboCup simulation league include [Stone and Veloso, 1999a] and [Pfeifer and Wainer, 2000].

Opponent Modelling

Opponent modelling and reasoning about other agents' goals is a key issue in adversarial multi-agent systems. It is a great advantage for an agent to be able to predict an opponent's intentions based on observations of previous actions. For example an agent may have observed that the left defender of the opposing team always passes the ball to the midfielder in front of him. This agent can move to intercept or block this pass whenever the ball goes to the left defender. Doing so could render the opposition's plan unusable, especially if a fixed passing policy is used. Also, general strategy recognition could lead to beneficial on-line formation changes in order to exploit tactical weaknesses in the opposing team. Initial work on opponent modelling is presented in [Riley, 1999].

2.3 The RoboCup Soccer Server

As mentioned earlier, the server [Noda et al., 1998] has been developed at the Electro-Technical Laboratory, Japan, and has been the basis for a number of successful international competitions. Experimental results reported in this thesis have been

conducted in several different versions of the simulator ranging from version 4 to the current version 6. This chapter describes the current version (i.e. version 6).

The server provides a complex and realistic domain for AI and MAS research. Unlike many other AI domains, the server embraces as many real-world complexities as possible [Stone, 1998]. In doing so, it integrates features from different existing systems as well as from human soccer players. For example the server's sensor and actuator noise models are motivated by typical robotic systems, while other characteristics, such as limited stamina and hearing distance, are motivated by human players.

While not as detailed as the server's user manual [Chen et al., 2001], this section defines all of the concepts and parameters that are important for the thesis. This description involves numerous operational parameters (used by the server), that will appear **in bold**. A brief description of these parameters together with their default values is given in table C.1 (in appendix C).

2.3.1 Overview of the Server

The RoboCup simulator can be seen as a standard client/server application. The simulator, acting as a server, provides the environment in which a number of agents operate. Agents, (also referred to as *clients* due to the client/server application context) connect to the server via UDP/IP (User Datagram Protocol/Internet Protocol) sockets. Each client program is responsible for controlling a single agent. More specifically, each client program acts as the brain of each agent, sending control commands (such as *kick* or *turn*) to the server. The server executes these control commands, simulates the movements of all of the objects (i.e. the agents, the ball, etc.), and updates the state of the environment accordingly. In addition, the server sends sensory information from that agent's perspective back to the client.

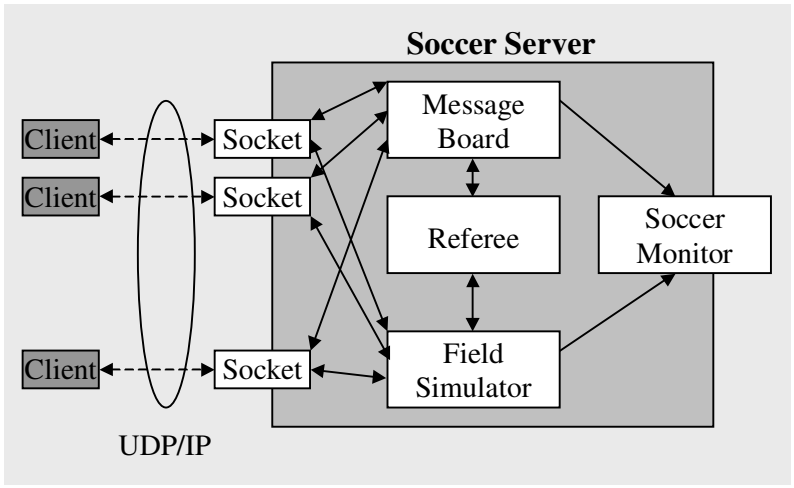


Figure 2.1: Overview of the server (based on similar figure in [Noda et al., 1998]).

When a game is to be played, two teams of eleven independently controlled agents connect to the server. In a similar manner to human soccer games, the objective of each team is to score more goals than the opposition in order to win. In other words, each team tries to direct the ball into the opposing team's goal, while preventing the ball from entering its own goal.

An overview of the server can be seen in figure 2.1. Its three main modules are the *message board*, the *referee* module, and the *field simulator*. The message board is responsible for handling all communication between the clients and the server. The referee enforces the rules of the game. In other words, the referee indicates changes in the play mode, such as when a goal is scored or when the ball goes out of bounds. The referee is also responsible for enforcing the offsides rule. Finally the field simulator is responsible for moving all the objects on the field. The position of each object is calculated based on control commands received for this object and also possible collisions that may occur between different objects.

The server also provides with a visualisation tool, which displays a graphical representation of the state of the environment. This module, also known as the *soccer*

monitor, includes information regarding the positions of the agents and the ball, the current score, current cycle number, and team names. A screen shot of the soccer monitor can be seen in figure 2.2. The status bar on the top provides information regarding the team names (Essex vs. Wizards), score (0 – 0), current play mode (kick_off_l) and the time cycle 20. The playing field together with the two goals (represented by the two black bars at the edges of the field), twenty-two agents, and the ball can be seen below the status bar. Each agent is represented as a two-halved circle. The light side is the side towards which the agent is facing. For example in figure 2.2 all twenty-two agents are facing the ball, which is in the middle of the field.

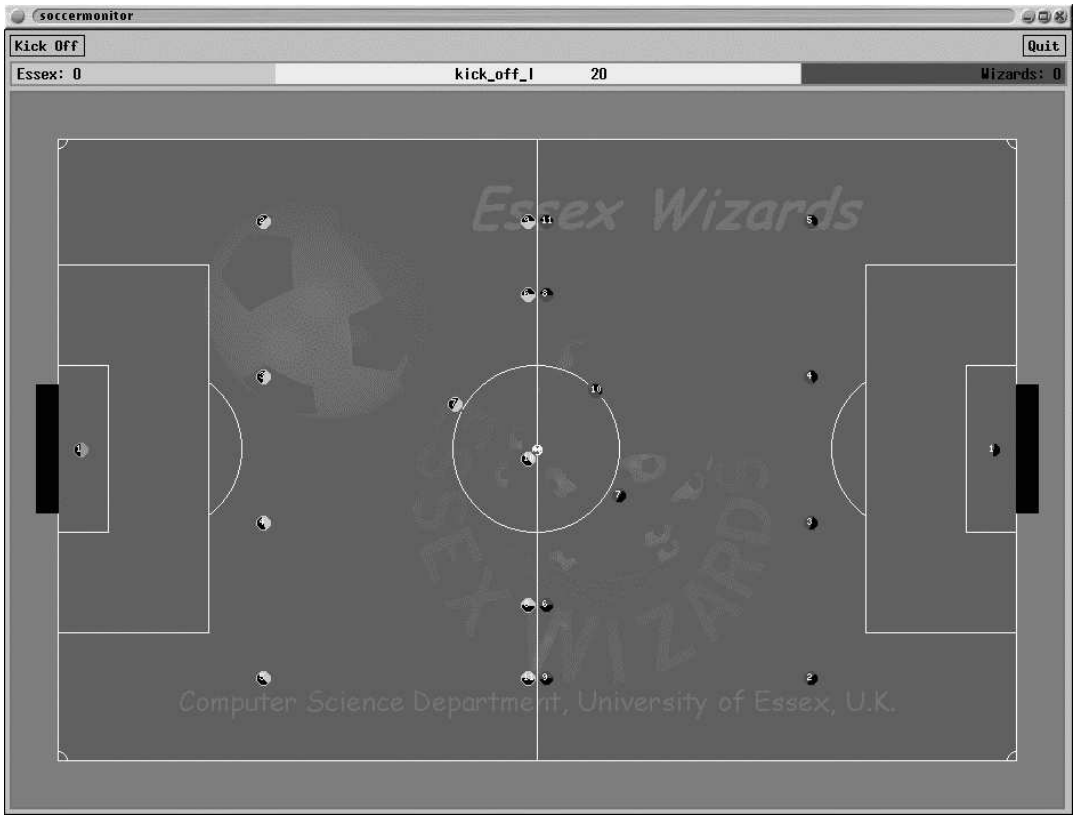


Figure 2.2: The soccer monitor.

The server provides a challenging testbed for MAS research due to the characteristics it combines. Some of these characteristics include:

- *Hidden state* – The limited field of view provided by the server leads to large amounts of hidden state.
- *Presence of both teammates and adversaries* – An agent can co-operate with its teammates in order to achieve its goal. However, there is another team of agents with a conflicting goal.
- *Noisy sensors and actuators* – The agent cannot perceive the environment exactly as it is, nor can it affect it exactly as intended.
- *Asynchronous sensing and acting rates* – The traditional AI paradigm of using perceptual input to trigger actions cannot be used in the server since it provides with independent sensing and acting rates.
- *Real-time decision making* – Each agent must make its decisions in real-time. In most cases a result of poorer quality but on time is preferred to a late highly accurate result.

Another characteristic of the server is that it uses a discrete action model. More specifically, it collects agent actions over the course of a fixed *simulation cycle* of length **simulator_step,** but only executes them and updates the environment at the end of the cycle [Stone, 1998]. If an agent fails to send an action request before the end of a cycle it will remain idle for that cycle. This can be a significant disadvantage in a real-time adversarial domain since opponents may gain an advantage. On the other hand if the agent sends more than one action, the server will only execute the first one it receives, which may lead to undesired results. (Actually the agents can send a number of *secondary* actions but are only allowed one *primary* action per cycle. A description of the primary and secondary action sets follows shortly.) When a **simulator_step** interval has elapsed, the server increments the simulated time counter by one and proceeds to the next cycle. Building agents that can act in real-time and handle asynchronous sensing and

acting, especially when sensing can occur at unpredictable intervals, is a challenging task. Agents must balance the need to act regularly and in real-time with the need to gather information regarding the environment.

An additional challenge is due to the communication channel used between the server and the clients. UDP/IP is an unreliable protocol, which means that an action request sent by a client is not guaranteed to reach the server. Therefore, agents must use future sensations in order to verify whether certain actions have been executed. Although in recent years better hardware reduces the amount of message loss, it still remains an important consideration when building a team for RoboCup competitions. Undetected action loss can lead to a number of problems especially during critical operations such as ball handling.

Apart from the challenges described above, the simulator captures several other real-world complexities that will become evident throughout the remainder of this chapter. All these properties make RoboCup a challenging domain with many similarities to real life environments. In addition, these characteristics are appropriate for the development of this thesis' contributions as presented in chapter 1.

2.3.2 The Simulated World

Field and Objects

As described in [Stone, 1998]:

> The simulator provides with a two-dimensional environment. There is no notion of height for any of the objects on the playing field. The field has dimensions **field_length** × **field_width** with the goals having a width of **goal_width**, all of

which are parameters defined in table C.1. There are up to 22 agents on the field at any time along with 1 ball, all of which are modelled as circles. There are also 55 visible markers (53 flags and 2 goals) distributed around the field. All 55 markers are illustrated (as small squares) in figure 2.3. Finally, there are 4 sidelines. The sidelines together with the markers are used for localisation, i.e. when an agent tries to estimate its exact position on the field. Note that the server provides each agent with visual information (i.e. what the agent can see), but it does not report the location of the agent on the field. Therefore the agent has to calculate its location based on noisy visual sensations.

The agents and the ball are all mobile objects. At the simulator cycle for time t, each object is in a specific position (p^t_x, p^t_y) and has a specific velocity (u^t_x, u^t_y). Each agent is also facing in a specific direction θ^t. These positions are maintained internally as floating point numbers, but agent sensations are only given to one decimal place.

[Stone, 1998].

Note that **field_length** = 105m and **field_width** = 68m, therefore the (x,y) co-ordinates of the agents and the ball used alone provide with:

$$^{105\times 68}C_{23} = {}^{7140}C_{23} = \frac{7140!}{23!\times(7140-23)!} = \frac{7140!}{23!\times 7117!}$$

possible configurations (or states) to arrange all agents and the ball on the play region. If the agents' velocities together with their body and neck facing directions are taken into account, the state space becomes even larger.

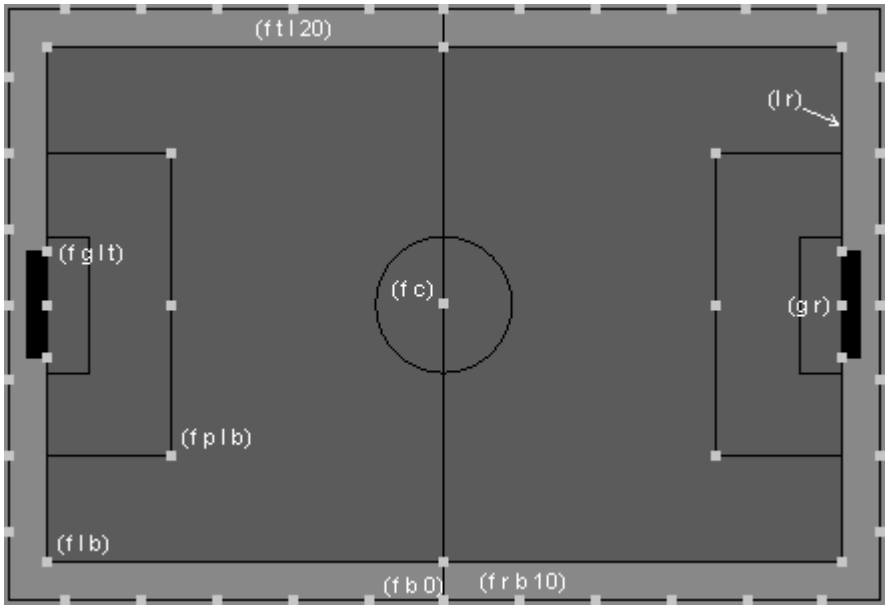


Figure 2.3: The locations of the server's 55 visible markers (illustrated as small squares).

Object Movement

As described in the user manual:

In each simulation step, the movement of each object is calculated in the following manner:

$$(u_x^{t+1}, u_y^{t+1}) = (u_x^t, u_y^t) + (a_x^t, a_y^t) \text{ : accelerate} \quad (2.1)$$

$$(p_x^{t+1}, p_y^{t+1}) = (p_x^t, p_y^t) + (u_x^{t+1}, u_y^{t+1}) \text{ : move}$$

$$(u_x^{t+1}, u_y^{t+1}) = decay \times (u_x^t, u_y^t) \text{ : decay speed}$$

$$(a_x^{t+1}, a_y^{t+1}) = (0,0) \text{ : reset acceleration}$$

where, decay is a parameter specified by **ball_decay** or **player_decay**. (a^t_x, a^t_y) is the acceleration of the object, which is derived from the *Power* parameter in *dash* commands (if the object is an agent) or *kick* commands (if the object is the ball) in the following manner:

$$(a^t_x, a^t_y) = Power \times (\cos(\theta^t), \sin(\theta^t))$$

If the object is an agent, its direction is calculated in the following manner:

$$\theta^t = \theta^t + Moment$$

where *Moment* is the parameter in *turn* commands. If the object is the ball, its direction is given by:

$$\theta^t_{ball} = \theta^t_{kicker} + Direction$$

where θ^t_{ball} and θ^t_{kicker} are the directions of the ball and kicking agent respectively, and *Direction* is the second parameter of the *kick* command.

Collisions

If at the end of the simulation cycle, two objects overlap, then the objects are moved backwards until they do not overlap. Following this, the velocities are multiplied by –0.1. Note that it is possible for the ball to go through an agent as long as the ball and the agent never overlap at the end of the cycle.

Noise and winds

The server adds evenly distributed probabilistic noise to all objects' movements. In particular noise is added into the equations given above as follows:

$$(u_x^{t+1}, u_y^{t+1}) = (u_x^t, u_y^t) + (a_x^t, a_y^t) + (r'_{r\max}, r'_{r\max})$$

where r'_{max} is a random number whose distribution is uniformed between the range [-max, max]. rmax is a parameter that depends on the velocity of the object as follows:

$$r\max = rand \cdot \left|(u_x^t, u_y^t)\right|$$

where *rand* is a parameter specified by **ball_rand** or **player_rand**. Noise is also added into the *Moment* parameter of a *turn* command as follows:

$$Moment = (1 + r'_{rand}) \cdot Moment$$

[Chen et al., 2001].

Stamina

The server prevents agents from constantly running at maximum velocity (**player_speed_max**) by assigning each agent a limited stamina [Stone, 1998]. A detailed description of the server's stamina model can be found in the user manual [Chen et al., 2001].

2.3.3 Client/Server Interface

The interface between the clients and the server can be broken down into 2 main components, namely, *agent perception* and *agent action*. *Agent perception* deals with the flow of data from the server to the clients, while *agent action* deals with the flow of data

from the clients to the server (figure 2.4). The detailed description that follows should provide with a better understanding of the agents' capabilities.

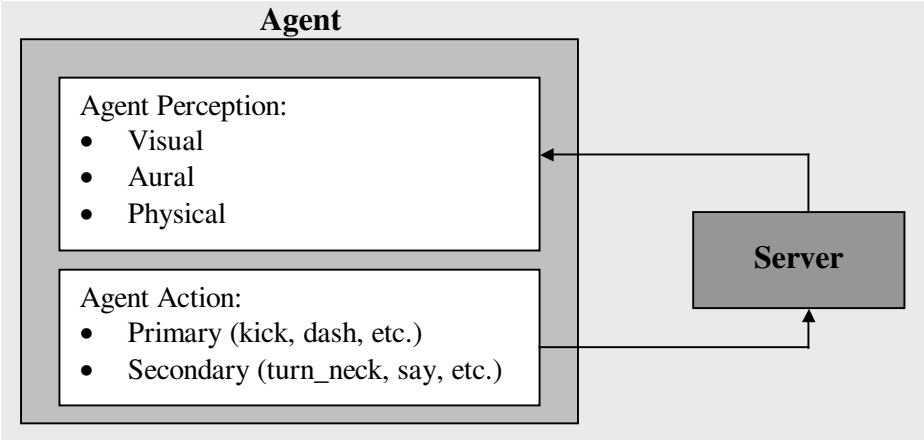


Figure 2.4: Client/Server interface.

2.3.3.1 Agent Perception

Table 2.1: Characteristics of sensory information.

Name	**When Received**	**Information Type**	**Limitations**
See	Every **send_step** msec (default)	Visual	Limited angle, precision decreases with distance
Hear	Instantaneously when a nearby agent (or the referee) speaks	Aural	Limited distance and frequency
Sense_Body	Every **sense_body_step** msec or upon request	Physical	None

Agents receive 3 types of sensory perceptions from the server, namely, visual, aural, and physical. This section presents the characteristics of the sensory information, which are summarised in table 2.1.

Aural Information

The server's communication paradigm models a crowded, low-bandwidth environment [Stone, 1998]. A detailed description of how the server handles communication between agents can be found in the user manual [Chen et al., 2001]. However, since this thesis deals with non-communicating MAS, the proposed implementation does not make use of the server's communication model.

Visual Information

As described in the user manual:

> Visual information arrives from the server in the following format:
>
> (see *Time Object_1_Info Object_2_Info ... Object_n_Info*)
>
> *Time*, as in the case of audio messages, indicates the current simulation cycle. What follows, is a list of all visible objects. Data regarding a particular object has the following format:
>
> (*Object_Name Distance Direction DistChng DirChng Body_Face_Dir Head_Face_Dir*)

OR (*Object_Name Distance Direction DistChng DirChng*)

OR (*Object_Name Distance Direction*)

OR (*Object_Name Direction*)

How much information is given for a particular object is determined based on how far the object is, and the quality of the field of view. Also, *Body_Face_Dir* and *Head_Face_Dir* are only applicable if the seen object is another agent.

Object_Name can be one of the following:

(p *Team_Name Uniform_Number Goalie*) – where p stands for player

(g [l|r]) – where g stands for goal, l stands for left, and r stands for right

(b) – where b stands for ball

(f c) – where f stands for flag and c stands for centre

(f [l|c|r] [t|b]) – where t stands for top and b stands for bottom

(f p [l|r] [t|c|b]) – where p stands for penalty area

(f g [l|r] [t|b]) – where g stands for goal flag

(f [l|r|t|b] 0)

(f [t|b] [l|r] [10|20|30|40|50]) – where 10, 20, etc. indicate distance offsets

(f [l|r] [t|b] [10|20|30])

(l [l|r|t|b]) – where l stands for line

(P) – where P stands for player

(G) – where G stands for goal

(B) – where B stands for ball

(F) – where F stands for flag

Distance, *Direction*, *DistChng*, *DirChng*, *Body_Face_Dir*, and *Head_Face_Dir* are calculated in the following way:

$$p_{rx} = p_{xt} - p_{xo}$$

$$p_{ry} = p_{yt} - p_{yo}$$

$$u_{rx} = u_{xt} - u_{xo}$$

$$u_{ry} = u_{yt} - u_{yo}$$

$$Distance = (p^2_{rx} + p^2_{ry})^{1/2}$$

$$Direction = \arctan(p_{ry}/p_{rx}) - \theta_0$$

$$e_{ry} = p_{rx} / Distance$$

$$e_{ry} = p_{ry} / Distance$$

$$DistChng = (u_{rx} \times e_{rx}) + (u_{ry} \times e_{ry})$$

$$DirChng = [(- (u_{rx} \times e_{ry}) + (u_{ry} \times e_{rx})) / Distance] \times (180 / \pi)$$

where (p_{rx}, p_{ry}) is the absolute position of the target object, (p_{xo}, p_{yo}) is the absolute position of the sensing agent, (u_{xt}, u_{yt}) is the absolute velocity of the target object, (u_{xo}, u_{yo}) is the absolute velocity of the sensing agent, and θ_0 is the absolute direction the sensing agent is facing. In addition, (p_{rx}, p_{ry}) and (u_{rx}, u_{ry}) are respectively the relative position and relative velocity of the target object, and (e_{rx}, e_{ry}) is the unit vector that is parallel to the vector of the relative position. As mentioned earlier, *Body_Face_Dir* and *Head_Face_Dir* are only included if the observed object is an agent, and is the body and head direction of the observed agent relative to the body and head direction of the observing agent. In other

words, if both agents have their bodies turned in the same direction, then *Body_Face_Dir* would be 0. The same applies for *Head_Face_dir*.

The remaining objects are best explained by example. The (g r) object indicates the centre of the right goal. (f c) is a virtual flag at the centre of the field. (f l b) is the flag at the lower left of the field. (f p l b) is the flag at the lower inside corner of the penalty box on the left side of the field. (f g l t) is the top goal flag of the left goal, i.e. the top goal post of the left goal. All flags with numerical identifiers are located 5 meters outside the playing field. For example (f t l 20) is 5 meters from the top sideline and 20 meters to the left of the centre line. In the same way, (f r b 10) is 5 meters right of the right sideline and 10 meters below the centre of the right goal. Also (f b 0) is 5 meters below the midpoint of the bottom sideline. In the case of line objects (e.g. (l r)), *Distance* is the distance to the point where the centre line of the agent's view crosses the line and *Direction* is the direction of the line. All the flags and lines used as examples in the paragraph above can be seen in figure 2.3.

[Chen et al., 2001].

Agents are also equipped with proximity sensors and can *feel* objects even if these objects are not within the view cone of the sensing agent. In other words, if an object is outside the agent's field of view but is within **feel_distance** meters from the agent, the agent receives partial information regarding that object. Felt objects are denoted with capital letters in the visual string, i.e. (P), (G), etc.

As described in [Stone, 1998]:

The frequency, range, and quality of visual information sent to a particular agent are governed by the integer server parameters **send_step** and **visible_angle**, and the agent parameters ***view_quality*** ∈ {*high*, *low*}, ***view_width*** ∈ {*narrow*, *normal*, *wide*}, and the floating point number ***view_angle***. An agent can control its ***view_quality*** and ***view_width*** via its actuators. By default, ***view_quality*** = *high*, ***view_width*** = *normal*, and ***view_angle*** = **visible_angle**. In that case the agent receives visual information every **send_step** msec.

However, the agent can trade off the frequency of the visual information against its quality and width via its ***view_mode*** = {***view_quality***, ***view_width***}. By using *low* ***view_quality***, in which case it receives only direction information but no distances to objects, the agent can receive visual information twice as frequently, i.e. every **send_step** / 2 msec. Similarly, the agent can receive visual information twice as frequently by switching to a *narrow* ***view_width***, in which case ***view_angle*** = **visible_angle** / 2. On the other hand, the agent can use a *wide* ***view_width*** (i.e. ***view_anlge*** = **visible_angle** × 2) at the expense of less frequent visual information (every **send_step** × 2 msec). With both ***view_width*** = *narrow* and ***view_quality*** = *low*, visual information arrives every **send_step** / 4 msec.

The meaning of the ***view_angle*** parameter is illustrated in figure 2.5. In this figure, the viewing agent is the one shown as two semi-circles. The light side is the side towards which the agent is facing. The black circles represent various objects in the environment. Only objets within ***view_angle***, and those within **feel_distance** of the viewing agent can be seen. Thus, objects *b* and *g* are not visible; all of the rest are. As object *f* is directly in front of the viewing agent, its

angle would be reported as 0 degrees. Object *e* would be reported as being at roughly –40 degrees, while object *d* is at roughly 20 degrees.

Also illustrated in figure 2.5, the amount of information describing an agent varies with how far away the agent is. For nearby agents, both the team and uniform number of the agent are reported. However, as distance increases, first the likelihood that the uniform number is visible decreases, and then even the team name may not be visible. It is assumed in the server that **unum_far_length** ≤ **unum_too_far_length** ≤ **team_far_length** ≤ **team_too_far_length**. Let an agent's distance be *Distance*. Then

- if *Distance* ≤ **unum_far_length**, then both the uniform number and team name are visible
- if **unum_far_length** < *Distance* < **unum_too_far_length**, then the team name is always visible, but the probability that the uniform number is visible decreases linearly from 1 to 0 as *Distance* increases
- if *Distance* ≥ **unum_too_far_length**, then the uniform number is not visible
- if *Distance* ≤ **team_far_length**, then the team name is visible
- if *Distance* < **team_far_length** < **team_too_far_length**, then the probability that the team name is visible decreases linearly from 1 to 0 as *Distance* increases
- if *Distance* ≥ **team_too_far_length**, then neither the team name nor the uniform number is visible

For example, in figure 2.5, assume that all of the labelled black circles are agents. Then agent *c* would be identified by both team name and uniform number;

agent *d* by team name, and with about a 50% chance, uniform number; agent *e* with about a 25% chance, just by team name, otherwise with neither; and agent *f* would be identified simply as an anonymous agent.

Finally, the precision of information regarding an object's distance decreases with its distance. In the case that an object in sight is a ball or an agent, the value of distance to the object is quantised in the following manner:

$$d' = Quantise(exp(Quantise(log(d), \textbf{quantise_step})), \textbf{quantise_step})$$

where d and d' are the exact distance and quantised distance respectively, and

$$Quantise(V, Q) = rint(V/Q) \times Q$$

This means that agents can not know the exact positions of very far objects. For example, when the distance is about 100.0 meters, the maximum noise is about 10.0 meters, while when the distance is less than 10.0 meters, the noise is less than 1.0 meter. In the case of flags and lines, the distance value is quantised in the following manner:

$$d' = Quantise(exp(Quantise(log(d), \textbf{quantise_step_l})), \textbf{quantise_step})$$

This visual paradigm creates a huge amount of hidden state for each individual agent. In addition, with the parameters identified in table C.1, by default, the agents receive visual information less frequently (every 150 msec) than they can

act (every 100 msec). Thus, unless they are to miss many action opportunities, the agents must sometimes choose more than one sequential action from a given state.

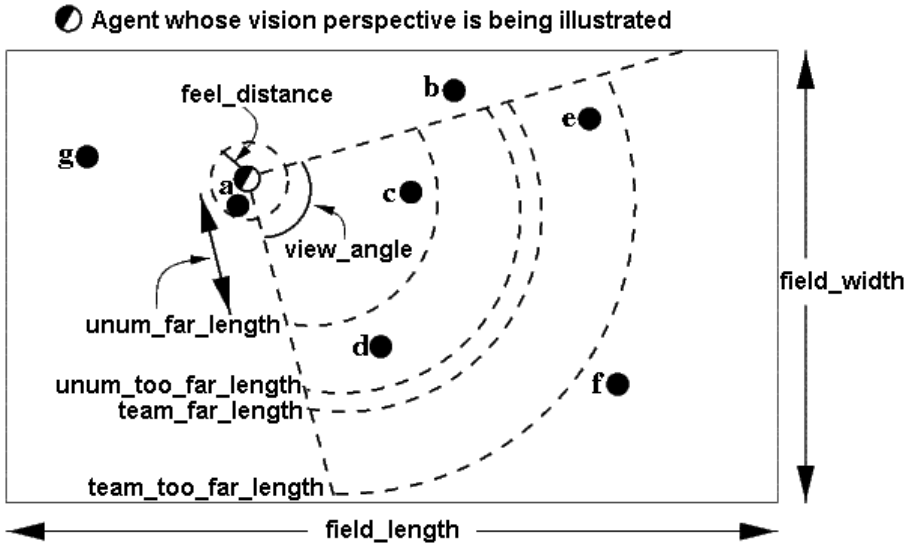


Figure 2.5: The ***view_angle*** parameter and its effect on visual sensations (figure originally appeared in [Stone, 1998]).

With these parameters, play occurs in *real-time*. In other words, agents must react to their sensory inputs at roughly the same speed as robotic or human soccer players. The agents developed for the Essex Wizards MAS have been able to act between 3 and 40 times per second. Human reactions have been measured in the 200-500 msec range [Woodworth, 1938].

[Stone, 1998].

Physical Information

Every **sense_body_step**, the server sends information regarding the *physical* status of an agent. This information is also available upon request by the agents. The format of the body sensor message is as follows:

(sense_body *Time*

(view_mode *view_quality view_width*)

(stamina *s_stamina s_effort*)

(speed *s_magnitude s_direction*)

(head_angle *h_direction*)

(kick *k_count*)

(dash *d_count*)

(turn *t_count*)

(say *s_count*)

(turn_neck *tn_count*)

(catch *c_count*)

(move *m_count*)

(change_view *cv_count*))

Time, as in the case of audio and visual data, represents the current simulation cycle. view_mode is as described above with *view_quality* being either high or low and *view_width* being one of narrow, normal, or wide. speed represents the agents velocity and reports its magnitude (*s_magnitude*) and direction (*s_direction*). head_angle is the relative direction of the agents head (with respect to its body). The remaining variables give the total number of commands of a particular type executed by the server. The agent can use these counters to determine action loss. All of the commands mentioned in the physical sensory input are described in the following section.

2.3.3.2 Agent Action

This section is based on the user manual [Chen et al., 2001] and [Stone, 1998].

An agent can send a number of action requests to the server. This section describes the characteristics of the different actions, which are summarised in table 2.2. The default values for the min/max variables used in the table above, can be found on table C.1, together with the rest of the server parameters.

One of the most important characteristics of each action is its class, i.e. whether it is a *primary* or a *secondary* action. From the set of all primary actions, the server will only execute 1 per cycle. If more that one request for a primary action is sent during the same cycle, the server will only execute the first one it receives and will ignore the rest. For example if an agent sends multiple *dash* requests during a cycle, the server will only execute the first it received. If an agent sends a *dash* followed by a *turn*, the server will again execute only the *dash* and ignore the *turn* (assuming the requests arrive at the server in the order they were sent). On the other hand the server will execute as many secondary requests as it receives. In other words, if an agent sends a *change_view*, a *sense_body*, and a *say* request, all during the same cycle, the server will execute them all. However, to avoid flooding the server, an agreement between the teams prohibits sending more than four requests per cycle (per agent).

Primary and secondary requests are not mutually exclusive. An agent can send both a primary and a number of secondary requests during the same cycle. As long as the total number of actions does not exceed four, and there is only one primary action in the set, the server will execute all of them. This is a significant design constraint during the development of a team. An agent that fails to operate under the timing constraints

specified by the server is guaranteed to under-perform. For example, if the agent fails to send an action request before the end of a cycle it will remain idle for that cycle. This can be a significant disadvantage in a real-time adversarial domain since opponents may gain an advantage. On the other hand if the agent sends more than one primary action, the server will only execute the first one it receives, which may lead to undesired results.

Table 2.2: The different actions available to agents.

Syntax	**Argument Meaning**	**Argument Range**	**Class**
catch(x)	direction to catch	[**minmoment**, **maxmoment**]	Primary
change_view(x, y)	view quality, view width	high/low, narrow/normal/high	Secondary
dash(x)	power to dash	[**minpower**, **maxpower**]	Primary
kick(x, y)	power to kick, direction to kick	[**minpower**, **maxpower**], [**minmoment**, **maxmoment**]	Primary
say(x)	message to be broadcasted	Less than **audio_msg_size**	Secondary
sense_body()			Secondary
turn(x)	angle to turn by	[**minmoment**, **maxmoment**]	Primary
turn_neck(x)	angle to turn neck by	[**minneckmoment**, **maxneckmoment**]	Secondary

The rest of this section briefly describes each of the actions shown in table 2.2. To ease the description, the functionality of each action is used to separate them into the following four categories:

1. Communication actions
2. Movement actions
3. Ball handling actions
4. Perception control actions

Communication Actions

Say(Message):

Agents can use the *say* command in order to broadcast messages to other agents. Messages can be at most **say_msg_size** bytes long and can only be heard by agents within a **audio_cut_dist** meter radius from the source. Since communication within the server is carried out via a single channel, both teammates and adversaries will hear messages sent by a given agent, as long as they are within hearing distance. *Say* requests sent to the server will be sent back to agents immediately (i.e. within the same cycle). The use of the *say* command is restricted by the limited hearing capacity of the agents as described earlier.

Movement Actions

Turn(Moment):

The *turn* command is used to change the agent's body direction. The argument of a *turn* command is the moment, which must be in the range of **minmoment** and **maxmoment**.

When an agent's velocity is 0, the moment specified in the *turn* command is equal to the angle the agent will turn. However, inertia makes it more difficult to turn when an agent is in motion. More specifically, the actual angle an agent will turn is computed as follows:

$$actual_angle = Moment / (1.0 + \textbf{\textit{inertia_moment}} \times agent_speed)$$

Dash(Power):

The *dash* command is used to accelerate an agent. Agents can only move in the direction their body is facing. The parameter of a *dash* command is the power, which must be in the range of **minpower** and **maxpower** (if the specified power of a *dash* command is negative, the agent moves backwards). Dashing efficiency is reduced for agents with low stamina as described previously. A *dash* command sets the acceleration in equation 2.1, after the specified power has been multiplied by **dash_power_rate**. This acceleration is valid for one simulation cycle and does not cause a sustained run over time. In order to keep running an agent must keep issuing dash commands every cycle.

Ball Handling Actions

Kick(Power, Moment):

The *kick* command is used to accelerate the ball in the same way that a *dash* command is used to accelerate an agent. The moment controls the direction of the kick and must be in the range of **minmoment** and **maxmoment**. The power of a kick command must be in the range of **minpower** and **maxpower**. The power is used to set the acceleration in equation 2.1, after it has been multiplied by the *kick_power_factor*, which is computed as follows:

kick_power_factor = ***kick_power_rate*** ** (1 – 0.25 * dir_diff / 180 –*

*0.25 * (dist_ball –* ***player_size*** *–* ***ball_size****) /* ***kickable_margin****)*

where *dir_diff* is the absolute value of the angle of the ball relative to the direction the agent's body is facing (i.e. if the ball is directly ahead, this would be 0) and *dist_ball* is the distance from the centre of the agent to the ball. According to this model, the strongest kick occurs when the ball is as close as possible to the agent and directly in front of it. The maximum power the agent can kick the ball with decreases as distance and/or angle increases. Note, that a pre-condition for a *kick* command to be successful is that the ball is within the agent's *kickable_area*, where:

kickable_area = ***ball_size*** + ***player_size*** + ***kickable_margin***

Also note that the server models kicks by vector addition. In other words, a kick accelerates the ball in a given direction, as opposed to setting the velocity of the ball. Therefore, multiple kicks on successive cycles can move the ball faster than a single kick.

Catch(Direction):

The goalkeeper is the only agent with the ability to issue catch commands. The goalkeeper can catch the ball if the ball is within the *catchable_area* and both the goalkeeper and ball are inside the penalty area. The specified direction must be in the range of **minmoment** and **maxmoment**. The *catchable_area* is a rectangle of size **catchable_area_w** × **catchable_area_l** with one edge on the agent and extending in the direction specified. If the ball is within *catchable_area* it will be caught with probability

catch_probability. However, if a *catch* command is unsuccessful (for example if the ball is outside *catchable_area* when the command is issued), it takes **catch_ban_cycle** cycles until another catch command can be used (*catch* commands during this time have simply no effect).

Perception Control Actions

Turn_Neck(Moment):

The *turn_neck* command is used to turn an agent's head, independently of its body. The angle of the head of the agent is used to form the viewing cone for that agent. Although the *turn* command changes the angle of the agent's body, the *turn_neck* command changes the agent's neck angle relative to the body. The minimum and maximum values for the relative angle of an agent's neck are **minmoment** and **maxmoment** respectively. The specified moment of a *turn_neck* command must be in the range of **minneckmoment** and **maxneckmoment**. Note that the neck angle is relative to the agent's body so if a *turn* command is issued, the direction of the viewing cone changes even if no *turn_neck* command is executed. Also note that *turn_neck* is a secondary action and can therefore be sent during the same cycle as a primary action such as *turn* or *dash*. Finally, *turn_neck* commands are not affected by momentum like *turn* commands are.

Change_View(Quality, Width):

The *change_view* command is used to adjust the quality and width of the visual information received by the agent. The quality parameter can be one of {high, low} while the width parameter can be one of {narrow, normal, high}. As described earlier, higher quality and greater width result to less frequent visual information.

Sense_Body():

The sense_body command is used to request physical information from the server. Although the server automatically transmits physical information every **sense_body_step** msec, an agent can also receive this information upon request.

2.3.4 Sample Client/Server Interaction

This section describes a simple interaction between a client and the server. The aim is to provide a better understanding of the client/server interface and present in more detail what is happening during a match. Figure 2.6 shows the initial and final positions of the fragment that is presented in this sample. More specifically, figure 2.6 (a) shows an agent (labelled *AGENT*) that is about to run to the ball and kick it towards the enemy goal. Figure 2.6 (b) shows the state of the world a few cycles after the kick, just before the ball enters the enemy goal. In these screen shots, the text has been added for reference purposes and is not part of the actual display. The data shown are from the perspective of the agent labelled *AGENT* and illustrate most of the key points explained earlier regarding perceptual input and actuator output.

Initially the agent receives the following string:

```
(sense_body 24 (view_mode high normal) (stamina 3500 1) (speed 0 -29) (head_angle 0) (kick 0) (dash 0) (turn 6) (say 0) (turn_neck 0))
```

The *sense_body* identifier is used to determine the type of sensory input, in this case, physical. Following is a number (i.e. 24) representing the current cycle of the game (i.e. the elapsed time in the game). The rest of the input contains information regarding various properties of the agent (e.g. its view mode, head angle etc.) together with the

number of times it sent a particular action (this agent has only sent 6 *turn* commands so far).

Following this, the agent sends a *turn* request by transmitting the following string:

```
(turn 25.00)
```

As explained earlier, the argument represents the angle to turn by, in this case 25 degrees. Notice how in the following *sense_body* input (i.e. the second RECEIVED tag of the full trace shown in page 43), the action count for *turn* commands has been increased to 7. This indicates that the *turn* request sent by the agent has been received successfully by the server. Also, the time stamp indicates that the server has advanced to cycle 25.

Since the server has entered a new cycle, another *primary* command can be send. Therefore, the agent sends a *dash* request (with power 100) and in the same cycle it also receives the following input:

```
(see 25 ((g r) 19.9 -22) ((f r b) 47.9 13) ((f p r c) 11 34 -0 0) ((f p r b) 31.2 34) ((f g r t) 16.9 -42 -0 0) ((f g r b) 24.5 -8 -0 0) ((f b r 30) 50.4 40) ((f b r 40) 50.4 29) ((f b r 50) 51.9 18) ((f r 0) 24 -28 -0 0) ((f r b 10) 30 -11) ((f r b 20) 37.7 0) ((f r b 30) 46.5 6) ((b) 7.4 0 0 0) ((p "Attackers" 2) 22.2 5 0 0 -165 -165) ((p "Attackers") 40.4 21) ((p "Defenders") 22.2 -9) ((p "Defenders") 40.4 14) ((p "Defenders" 3) 22.2 20 0 0 -95 -95) ((p "Defenders" 4) 13.5 -40 0 0 114 114) ((P) 2 171) ((l r) 29.4 34))
```

The *see* identifier denotes visual data. As in the case of *sense_body* strings, the time stamp after the identifier indicates the current cycle, in this case 25. What follows is a list of all the objects that are within the field of view of the agent. For example ((g r) 19.9 –22) indicates that the right goal is 19.9 meters away at –22 degrees (in other words, 22 degrees anti-clockwise relative to the agents facing direction). Following, is a list of 12 visible flags, including penalty flags (e.g. ((f p r b) 31.2 34)), goal flags (e.g. ((f g r t) 16.9

–42 –0 0)), inner (e.g. ((f r b) 47.9 13)) and outer flags (e.g. ((f b r 40) 50.4 29)). Recall that the field has a total of 53 flags, but only 12 are visible to the agent at this time. The goal and flags used as examples above are also shown in figure 2.6 (a). For simplicity, the rest of the flags are ignored. A complete list of all visible markers used by the server (i.e. all flags and the two goals) can be seen in figure 2.3. After the flags, the visual string indicates the position of the ball, ((b) 7.4 0 0 0), which is directly ahead of the agent (i.e. at 0 degrees) 7.4 meters away. The last two digits indicate velocity information for the ball. In this case the ball is stationary. Following is a list of visible agents. For example the teammate with uniform number 2 is denoted as ((p "Attackers" 2) 22.2 5 0 0 –165 –165). This agent is a teammate because the team name (i.e. *Attackers*) matches the viewing agent's teamname. This agent is 22.2 meters away, 5 degrees clockwise. The following two digits indicate that it is stationary and the last two indicate the relative neck and body facing direction. It is worth mentioning that teammate number 3 is 40.4 meters away and therefore its uniform number if not visible to the viewing agent. Also agent number 5 is outside the field of view but within **feel_distance** of the agent. It is therefore denoted as ((P) 2 171), i.e. 2 meters away and 171 degrees clockwise. Note, in this case the viewing agent does not get uniform number or team name information. Finally the agent gets information regarding the visible lines. In this case only the right line is visible, shown as ((l r) 29.4 34).

Finally an example of an audio input can be seen from:

```
(hear 45 referee goal_l_1)
```

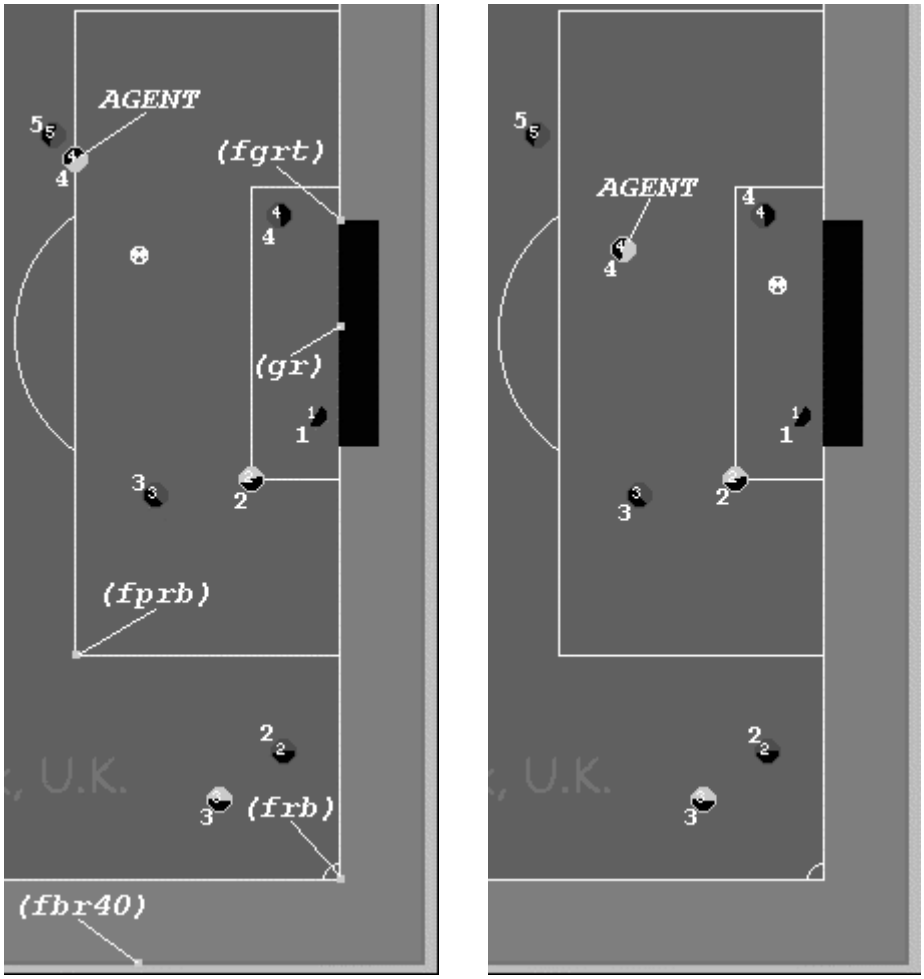
The *hear* identifier denotes audio input. As in the case of physical and visual sensations, the number that follows indicates the current cycle, in this case 45. Following is the *referee* identifier indicating the message is from the referee. Note that in the case of

another agent, such an identifier is not given. Instead this field is used to indicate the direction of the source. The remained of the input contains the message from the referee, in this case goal_l_1, indicating that the left team has scored its first goal.

A number of interesting observations can be made for this simple interaction. For example, due to the limited field of view of the agent, certain objects start to disappear from the visual string as the agent approaches the ball. For example opponent 4 is visible on cycle 26 (shown as ((p "Defenders" 4) 13.5 -42 -0.27 -0.8 114 114)) but is not shown on the visual string for cycle 28. Notice how on cycle 26 it's relative direction was –42 degrees (i.e. very close to the limit of the view cone, which spans from –45 to 45 degrees given that the agent's ***view_width*** is *normal*). In addition, the noise of the sensory input and actuator output can also be seen. For example, in the first visual string (i.e. cycle 26) the ball is directly ahead of the agent (i.e. 0 degrees). Although the agent only runs towards the ball and does not issue any *turn* or *turn_neck* commands, on cycle 31 the ball is 1 degree clockwise of the agent.

2.4 Summary

In short, this chapter described the main challenges of the RoboCup domain together with the key ideas behind the Robot World Cup Initiative. A detailed description of the server was also given. The characteristics of the server make it a challenging and realistic testbed for MAS research. Note that although the server provides with a communication model, this thesis contributes techniques that require no use of communication between teammates.



(a) (b)

Figure 2.6: Screen shots representing the initial (a)
and final (b) stages of the sample interaction.
(a) *AGENT* about to run to the ball and kick it towards the enemy goal.
(b) A few cycles after the kick, just before the ball enters the enemy goal.

BEGIN TRACE:

```
RECEIVED <- (sense_body 24 (view_mode high normal) (stamina 3500 1) (speed 0 -29)
(head_angle 0) (kick 0) (dash 0) (turn 6) (say 0) (turn_neck 0))
SENDING -> (turn 25.00)
RECEIVED <- (sense_body 25 (view_mode high normal) (stamina 3500 1) (speed 0 -55)
(head_angle 0) (kick 0) (dash 0) (turn 7) (say 0) (turn_neck 0))
SENDING -> (dash 100.00)
RECEIVED <- (see 25 ((g r) 19.9 -22) ((f r b) 47.9 13) ((f p r c) 11 34 -0 0) ((f p r b)
31.2 34) ((f g r t) 16.9 -42 -0 0) ((f g r b) 24.5 -8 -0 0) ((f b r 30) 50.4 40) ((f b r
40) 50.4 29) ((f b r 50) 51.9 18) ((f r 0) 24 -28 -0 0) ((f r b 10) 30 -11) ((f r b 20)
```

37.7 0) ((f r b 30) 46.5 6) ((b) 7.4 0 0 0) ((p "Attackers" 2) 22.2 5 0 0 −165 −165) ((p "Attackers") 40.4 21) ((p "Defenders") 22.2 −9) ((p "Defenders") 40.4 14) ((p "Defenders" 3) 22.2 20 0 0 −95 −95) ((p "Defenders" 4) 13.5 −40 0 0 114 114) ((P) 2 171) ((l r) 29.4 34))

RECEIVED <- (sense_body 26 (view_mode high normal) (stamina 3435 1) (speed 0.24 0) (head_angle 0) (kick 0) (dash 1) (turn 7) (say 0) (turn_neck 0))

SENDING -> (dash 100.00)

RECEIVED <- (see 26 ((g r) 19.3 −22) ((f r b) 47.5 14) ((f p r c) 10.5 35 −0.21 0.8) ((f p r b) 30.6 34) ((f g r t) 16.6 −43 −0.332 −0.6) ((f g r b) 23.8 −8 −0.476 −0.1) ((f b r 30) 49.9 41) ((f b r 40) 49.4 29) ((f b r 50) 51.4 18) ((f r 0) 23.6 −29 −0 −0.3) ((f r b 10) 29.4 −11) ((f r b 20) 37 0) ((f r b 30) 45.6 6) ((b) 6.7 0 −0.268 −0) ((p "Attackers" 2) 22.2 5 −0.444 0 −165 −165) ((p "Attackers") 40.4 21) ((p "Defenders") 22.2 −9) ((p "Defenders") 40.4 14) ((p "Defenders" 3) 20.1 21 −0.402 0.2 −95 −95) ((p "Defenders" 4) 13.5 −42 −0.27 −0.8 114 114) ((P) 2.5 173) ((l r) 28.8 34))

RECEIVED <- (sense_body 27 (view_mode high normal) (stamina 3370 1) (speed 0.37 0) (head_angle 0) (kick 0) (dash 2) (turn 7) (say 0) (turn_neck 0))

SENDING -> (dash 100.00)

RECEIVED <- (sense_body 28 (view_mode high normal) (stamina 3305 1) (speed 0.37 −2) (head_angle 0) (kick 0) (dash 3) (turn 7) (say 0) (turn_neck 0))

SENDING -> (dash 100.00)

RECEIVED <- (see 28 ((g r) 17.6 −25) ((f r b) 45.6 14) ((f p r c) 9.1 42 −0.182 1.7) ((f p r b) 29.1 36 −0 0.5) ((f g r b) 22 −9 −0.44 −0.1) ((f b r 30) 48.4 42) ((f b r 40) 47.9 30) ((f b r 50) 49.4 19) ((f r 0) 22 −31 −0.44 −0.5) ((f r b 10) 27.7 −12) ((f r b 20) 35.2 0) ((f r b 30) 43.8 6) ((b) 5 0 −0.4 0.3) ((p "Attackers" 2) 20.1 5 −0.402 0.2 −165 −165) ((p "Attackers") 40.4 22) ((p "Defenders" 1 goalie) 20.1 −10 −0.402 −0.1 154 154) ((p "Defenders") 36.6 15) ((p "Defenders" 3) 20.1 23 −0.402 0.5 −95 −95) ((l r) 26.8 34))

RECEIVED <- (sense_body 29 (view_mode high normal) (stamina 3240 1) (speed 0.36 1) (head_angle 0) (kick 0) (dash 4) (turn 7) (say 0) (turn_neck 0))

SENDING -> (dash 100.00)

RECEIVED <- (see 29 ((g r) 16.8 −26) ((f r b) 44.7 14) ((f p r b) 28.5 37) ((f g r b) 21.1 −9 −0.422 −0.2) ((f b r 30) 47.9 43) ((f b r 40) 47.5 31) ((f b r 50) 48.9 19) ((f r 0) 21.3 −33 −0.426 −0.6) ((f r b 10) 26.8 −13 −0.536 −0.2) ((f r b 20) 34.5 0) ((f r b 30) 42.9 6) ((b) 3.7 0 −0.37 −0) ((p "Attackers" 2) 20.1 6 −0.402 0.1 −165 −165) ((p "Attackers") 36.6 23) ((p "Defenders" 1 goalie) 18.2 −10 −0.364 −0.2 154 154) ((p "Defenders") 36.6 16) ((p "Defenders" 3) 18.2 24 −0.364 0.4 −95 −95) ((l r) 26 34))

RECEIVED <- (sense_body 30 (view_mode high normal) (stamina 3175 1) (speed 0.37 4) (head_angle 0) (kick 0) (dash 5) (turn 7) (say 0) (turn_neck 0))

SENDING -> (dash 100.00)

SENDING -> (change_view narrow high)

```
RECEIVED <- (see 30 ((f r b) 43.8 15) ((f g r b) 20.3 -10 -0.406 -0.3) ((f b r 50) 47.9
19) ((f r b 10) 25.8 -13 -0.516 -0.3) ((f r b 20) 33.4 0) ((f r b 30) 42.1 6) ((b) 3 0 -
0.36 -0.6) ((p "Attackers" 2) 18.2 6 -0.364 0 -165 -165) ((p "Defenders" 1 goalie) 18.2 -
11 -0.364 -0.3 154 154) ((p "Defenders") 36.6 16) ((l r) 25.3 34))
RECEIVED <- (sense_body 31 (view_mode high narrow) (stamina 3110 1) (speed 0.39 -3)
(head_angle 0) (kick 0) (dash 6) (turn 7) (say 0) (turn_neck 0))
RECEIVED <- (see 31 ((f r b) 42.9 15) ((f g r b) 19.3 -10 -0.386 -0.1) ((f b r 50) 47 20)
((f r b 10) 25 -14) ((f r b 20) 32.5 0) ((f r b 30) 41.3 7) ((b) 2 1 -0.4 1.1) ((p
"Attackers" 2) 18.2 6 -0.364 0.2 -165 -165) ((p "Defenders" 1 goalie) 16.4 -11 -0.328 -0.2
154 154) ((p "Defenders") 33.1 16) ((l r) 24.3 34))
SENDING -> (dash 100.00)
SENDING -> (turn_neck 3.80)
SENDING -> (change_view narrow high)
.
.
.
RECEIVED <- (see 39 ((g r) 13.6 10) ((f g r t) 12.6 -20 -0 -0) ((f r 0) 18.4 4 -0 -0) ((b)
1 -1 1.36 1.5) ((l r) 12.8 77))
SENDING -> (kick 100.00 -85.59)
RECEIVED <- (sense_body 40 (view_mode high narrow) (stamina 3325 1) (speed 0 34)
(head_angle -87) (kick 7) (dash 7) (turn 8) (say 0) (turn_neck 7))
SENDING -> (turn -86.63)
SENDING -> (turn_neck 87.00)
RECEIVED <- (see 40 ((g r) 13.6 10) ((f g r t) 12.6 -20 -0 -0) ((f r 0) 18.4 4 -0 -0) ((b)
3.3 0 2.178 0.4) ((l r) 12.8 77))
RECEIVED <- (sense_body 41 (view_mode high narrow) (stamina 2500 1) (speed 0 34)
(head_angle 0) (kick 7) (dash 7) (turn 9) (say 0) (turn_neck 8))
RECEIVED <- (see 41 ((g r) 13.6 10) ((f g r t) 12.6 -20 -0 -0) ((f r 0) 18.4 3 -0 -0) ((b)
6 0 2.28 0.1) ((l r) 12.8 76))
RECEIVED <- (sense_body 42 (view_mode high narrow) (stamina 2500 1) (speed 0 30)
(head_angle 0) (kick 7) (dash 7) (turn 9) (say 0) (turn_neck 8))
RECEIVED <- (see 42 ((g r) 13.6 10) ((f g r t) 12.6 -20 -0 -0) ((f r 0) 18.4 3 -0 -0) ((b)
8.2 0 2.132 -0.1) ((l r) 12.8 76))
RECEIVED <- (sense_body 43 (view_mode high narrow) (stamina 2500 1) (speed 0 25)
(head_angle 0) (kick 7) (dash 7) (turn 9) (say 0) (turn_neck 8))
RECEIVED <- (see 43 ((g r) 13.6 10) ((f g r t) 12.6 -20 -0 -0) ((f r 0) 18.4 3 -0 -0) ((b)
10 0 1.8 -0.2) ((l r) 12.8 76))
RECEIVED <- (sense_body 44 (view_mode high narrow) (stamina 2500 1) (speed 0 20)
(head_angle 0) (kick 7) (dash 7) (turn 9) (say 0) (turn_neck 8))
```

```
RECEIVED <- (see 44 ((g r) 13.6 10) ((f g r t) 12.6 -20 -0 -0) ((f r 0) 18.4 3 -0 -0) ((b)
12.2 0 1.708 -0.2) ((l r) 12.8 76))
RECEIVED <- (hear 45 referee goal_l_1)
```

END TRACE.

Chapter 3

Real-Time Multi-Threaded Agents

3.1 Overview

An important part in the development of any software project is its design phase. During the design phase, requirements are transformed into definitions of software components and their interfaces. The aim of this chapter is to present the architectural design of the Essex Wizards multi-agent system (MAS), together with some of the core features of the proposed approach.

One of the main issues in the development of MAS is the specification of the agent's architecture. Agent architectures describe the functional components of an agent and also how these components work together. There are numerous agent paradigms including reactive agents, deliberative agents, interactive agents, and layered agents. In order to select an acceptable framework, one must consider the requirements of the environment the agent is expected to operate in. The following section presents an analysis of the requirements of the RoboCup domain together with the proposed agent architecture.

Another important part of the proposed approach is the multi-threaded implementation of the *sense-think-act* cycle within the agent. Most of the existing implementations have focused on serial single-threaded approaches. It will be shown that such implementations are not suitable given the real-time requirements of the RoboCup domain. Experimental

results will demonstrate the superiority of multi-threaded agents in terms of performance and scalability.

Finally, this chapter demonstrates how the use of design patterns can aid the development of MAS. It has been shown [Gamma et al., 1995] that design patterns can help the potential developer deal with a number of problems including decomposing a system into objects, determining their granularity, and specifying their interfaces and implementation. Although a number of design patterns have been used in the development of the Essex Wizards MAS, the focus will be on a multi-threaded mediator pattern. This is primarily due to the importance of this component in the proposed architecture and the extensions that were introduced for multi-threaded support. Parts of this chapter have originally appeared in [Kostiadis and Hu, 2000] and [Kostiadis et al., 2000].

3.2 Agent Architecture

3.2.1 Requirements

Given the nature of the RoboCup domain, the response times of an agent become an important factor since the server operates in 100ms intervals for executing actions and 150ms intervals (under default settings) for providing visual sensory data. In addition, auditory sensory data can be received at completely random intervals. Therefore it is vital that each agent has bounded response times. If an action is not generated within 100ms, then the agent will remain idle for that cycle and opponents (that did act during that same cycle) might gain an advantage. On the other hand, if numerous primary actions are generated per cycle, the server will only execute one of them (normally the first to arrive

at the server), which might result to undesired behaviour. An additional constraint is that Unix is not a true real-time system and therefore real-time performance and response times can only be guaranteed up to a certain resolution [Stevens, 1998].

A real-time system is any system in which the time that output is produced is significant [Burns and Wellings, 1997]. This is true for RoboCup agents since they only have 100ms to generate an action for any given cycle. Hence, under the definition given above, a RoboCup agent can be classified as a real-time system, or a *real-time agent*.

Real-Time Requirements

The correct behaviour of non real-time agents is based on the *logical correctness* of the result. On the other hand, real-time agents require both the logical correctness of the result and its *timing correctness*. The definition of logical correctness (also known as functional correctness) depends on how a system is viewed abstractly. For example in systems exhibiting purely sequential behaviour the logical correctness is expressed in terms of correct inputs and outputs. Timing correctness can be viewed as an extension to logical correctness that includes a new dimension, namely, time.

These two notions of correctness may be traded off against each other. The result is two broad categories of real-time agents: *hard real-time agents* and *soft real-time agents* [Nissanke, 1997]. For hard real-time agents, timing correctness is of critical importance and should never be sacrificed for other gains. As an example consider a software agent controlling an aircraft. Failure to check the altitude at prescribed time intervals may have catastrophic consequences. In this case a result of poorer quality, but on time, is preferred to a late highly accurate result. On the other hand, for soft real-time agents, timing correctness is important but not critical. An occasional failure to meet a deadline should not result in serious consequences. Soft real-time agents perform tasks as fast as possible,

or as timely as possible, but their timing correctness may be sacrificed under special circumstances such as peak demands on the processor or communication medium.

Another distinction between real-time agents is whether their responses are based on *events* or *clock* times. This classification is of particular importance at the implementation level. *Event-triggered agents* respond to external stimuli by first detecting various events (also known as *signals* or *interrupts*) and then generating the appropriate actions dynamically. *Time-triggered agents* operate in accordance with clock times as shown by an integrated independent clock. In other words, clock pulses are treated as signals that generate certain actions. An appropriate action is selected based on the current state of the environment as perceived by the agent.

In order to build an efficient team of agents, the properties of the environment must be taken into account. Recall that the server operates in 100ms cycles. If an agent fails to meet this deadline for a given cycle, it will remain idle for that cycle. Competition experience has shown that cycle loss has been the primary reason for poor performance. Even teams with good high-level approaches under-performed due to slow response times. It is therefore required for RoboCup agents to be *hard real-time agents*. Another property of the server (from versions 5.00 and higher) is that cycle change information can be obtained from sensory data. However, since UDP/IP (User Datagram Protocol/Internet Protocol) communication is unreliable, agents cannot depend on sensory input to trigger actions. Therefore the agents can use sensory information as an event to trigger actions provided that a safety mechanism exists for when the sensory information is delayed or even lost. Hence, it is required that the agents are a *hybrid* of event-triggered and time-triggered agents.

Operational Requirements

As explained previously, the RoboCup simulation league is a client/server application in which each client communicates with the server via a UDP/IP socket. The server is responsible for executing requests from each client and updating the environment. At regular time intervals it broadcasts visual information to all clients depending on their position, the quality and size of their field of view, and their facing direction on the field. Moreover, the server sends auditory information to various clients at random time intervals.

After processing the sensory data, the clients respond by sending action requests to the server from a set of primitive actions available to them (e.g. kick, dash, turn, etc.). Clients are allowed to send only one primary action per cycle (100ms). If no action is sent within this cycle, the client will remain idle. If more than one primary action is send during the same cycle, the server executes only one (normally the first action to arrive at the server), which may produce undesired results. The server updates the state of the environment by executing all the requests by each agent for a given cycle.

It is important to mention that UDP/IP sockets have a limited receive buffer. Messages arriving on a UDP/IP socket will be queued until the receiving buffer is full in which case additional messages will be discarded. A client that fails to retrieve messages at the rate that they arrive is in danger of loosing data or receiving older information from the server, since newer packets will be further back in the queue. This will cause the client to create the wrong representation about the current state of the environment, leading to undesired results when the wrong actions are executed.

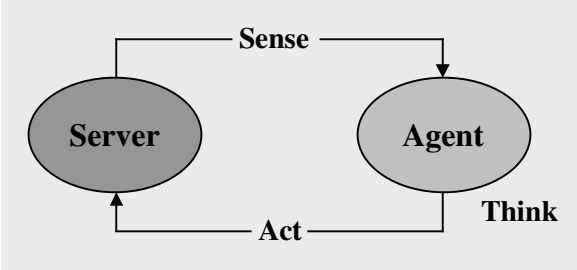


Figure 3.1: Agent's main loop.

The term *client*, used in the client/server application context above, is the real-time agent to be built. For each cycle, the agent receives data from the server (if new data is available), processes this data, and produces an action. A very basic model of an agent's loop can be seen in figure 3.1. When new data is available, the agent should *receive* the data and update the current state of the environment. It should then *think* and *send* an action to the server before the end of the cycle. For the agent to be efficient all of the following conditions need to be satisfied:

- The agent must be able to receive the newest sensory data that arrives on the socket as quickly as possible, and it should not let data queue up. This enables the agent to have the most recent representation of the environment, and hence execute the most appropriate action.
- The agent must be able to send the action requests to the server within the predefined time intervals, i.e. only one primary action per cycle. If more than one primary action is sent per cycle, the server will only execute the first one it received leading to undesired behaviour. On the other hand if the agent misses a cycle, opponents might gain an advantage.

- The agent must allocate the maximum amount of resources to the thinking process and minimise the time needed for communication with the server (i.e. sending and receiving data).

Given the frequency of the message exchange and the timing constraints, building an agent that will satisfy the conditions described above becomes a challenging task. Recall that an agent can receive a number of sensory inputs per cycle (e.g. visual, auditory, etc.) and can send a number of actions (one primary and a number of secondary actions). In addition, recall that all the client/server communication in the RoboCup domain is carried out via UDP/IP sockets. There are a number of ways the agent can handle I/O (Input/Output) operations on these sockets. Therefore, before introducing the agent architecture a brief description of the Unix I/O models is required.

3.2.2 Unix I/O Models

Unix provides five I/O models: blocking I/O, non-blocking I/O, I/O multiplexing, signal-driven I/O, and asynchronous I/O. Choosing the right model is an important part of the design process as it directly affects the performance of the agent. For the sake of completeness each I/O model is briefly introduced here. A more detailed description can be found in [Stevens, 1998].

- *Blocking* I/O mode is the default mode for all sockets under Unix, which means that a call to request data from a socket will not return until data is available. In other words, the calling process will be put to sleep until new data arrives or an error occurs.

- *Non-blocking* I/O mode avoids putting the process to sleep. If data is not immediately available, the kernel returns an error message, such as EWOULDBLOCK. The calling process will not be blocked and data can be requested at a later stage.
- *Multiplexing* I/O mode is achieved by using either the *select* or *poll* system calls. This method will block in one of these two calls rather than blocking at the actual I/O system call. When *select* returns, the socket is readable and the *recvfrom* function is called to copy the data into the application buffer.
- *Signal-driven* I/O mode enables the kernel to notify the process with a SIGIO signal when a descriptor is ready. To read the data from the socket the process needs to establish a signal handler for the SIGIO signal. To send actions to the server, an interval timer that generates a SIGALRM signal (when it expires) can be used. By setting the timer to 100ms (i.e. the duration of a cycle) and using a handler that sends actions to the server, the response times of an agent can be guaranteed within certain limits.
- *Asynchronous* I/O mode enables the kernel to notify the process when the entire I/O operation is complete. The difference with the signal-driven I/O is that the kernel tells the process when an I/O operation is complete rather than an operation can be initiated. Asynchronous I/O is not used very often primarily because it is not widely supported.

3.2.3 The Essex Wizards Architecture

Given this variety of I/O models supported under Unix, choosing the right one heavily depends upon the inner structure of the agent. In this section a novel architecture for building RoboCup agents is presented. The proposed architecture consists of 5 different

modules as shown in figure 3.2, namely the agent's *Sensors*, a set of *Behaviours*, the *Actuators*, a *World-Model* and a set of *Parameters*. The dashed arrows between the agent and the server in this diagram represent the communication links. The server sends information received by the agent's *Sensors* and the agent sends actions back to the server through its *Actuators*. The solid arrows represent dependence relationships between the modules. For example, the output produced by the *Actuators* is directly dependent upon the input it will receive from the *Behaviours*. All modules within the agent apart from the *Actuators* affect the *Behaviours*, which is the core of the decision making process of the agent.

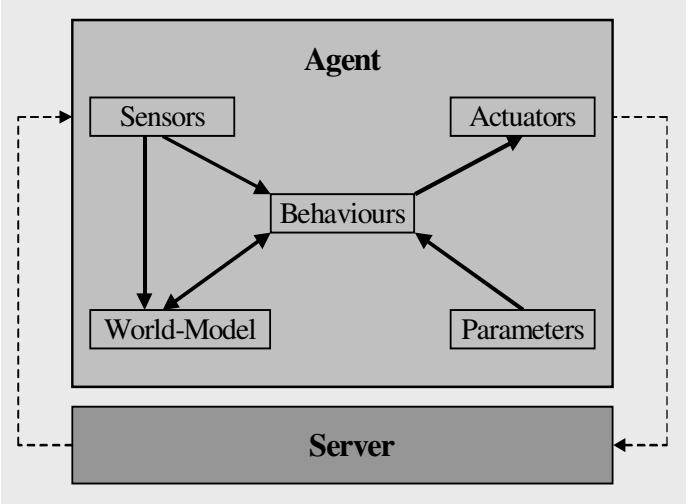


Figure 3.2: The Essex Wizards architecture.

Agent architectures normally describe the functional components within an agent and also how these components work together [Muller, 1998]. Such a description for the components of the Essex Wizards architecture is given below.

Sensors

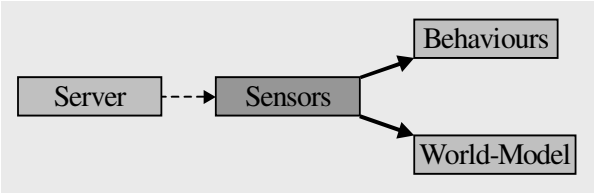


Figure 3.3: Sensors I/O.

The *Sensors* are responsible for receiving and analysing all the information transmitted by the server (e.g. visual data, audio data, etc.). After receiving sensory data for each cycle, the agent tries to create a representation of the environment. This representation is then used to generate the appropriate actions through the *Behaviours*. As can be seen from figure 3.2, *Sensors* affect the *Behaviours* of the agent both directly and indirectly (through the *World-Model*). This is due to the nature of the sensory data provided by the server. In the case of audio data, the server provides the agent with information regarding the source (i.e. referee or another agent), direction, and content of a message. Therefore, audio sensory data can be directly used by *Behaviours*. In the case of visual data, the server provides the agent only with partial information due to the limited field of view that each agent has. In most cases this is not sufficient in order to determine the best course of action. Therefore, all visual data are passed to the *World-Model* and are used to maintain a probabilistic representation of the whole environment. This is achieved by combining recent visual data together with visual data from previous cycles.

Behaviours

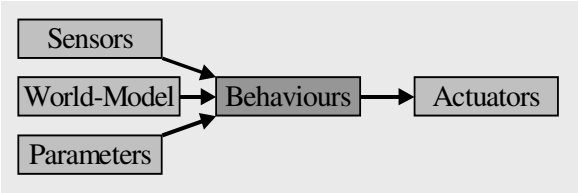


Figure 3.4: Behaviours I/O.

The *Behaviours* module is the most important module within the agent. As can be seen from figure 3.4, the *Behaviours* module is responsible for generating actions according to the current state of the environment. The state of the environment is determined using all the modules within the agent apart from the *Actuators*. Normally the agent needs to gather information from all other modules before an action is generated, including data regarding the agent's position and role in the team, the current formation, the position of the ball and other agents, the current play mode and so on. *Behaviours* process all this information and determine the best course of action. Unfortunately, given the complexity of the RoboCup domain, a direct mapping from states to actions is not feasible. This is mainly due to the extremely large size of the state space. Even if rules could be implemented that map each state to an optimal action, searching through these rules would take a considerable amount of time. Recall that an agent has only 100ms in which it has to receive and analyse sensory data, process that data and generate an appropriate action. Improving the efficiency of the decision making process of an agent is one of the main contributions of this thesis. A detailed analysis of the *Behaviours* module and the decision making process of the agent is given in subsequent chapters.

Actuators



Figure 3.5: Actuators I/O.

Actuators are responsible for sending actions to the server. Internally the *Actuators* are equipped with a receive buffer that can hold a number of actions. During the decision making process the *Behaviours* fill this buffer with action requests including both primary (e.g. kick, turn, etc.) and secondary (e.g. say, turn neck, etc.) actions. A trigger is used that causes the *Actuators* to initiate the sending of actions to the server. As mentioned earlier the agents are a hybrid of event-triggered and time-triggered agents. The event that is used is the *entering to a new cycle*, which is normally detected by checking the timestamp of sensory data received by the server. In addition, as a safety mechanism, the agents have a clock that can also trigger the *Actuators*. This clock is set to a time interval that is slightly larger than the duration of a cycle. Every time the agent enters a new cycle the clock is reset. However, if sensory data with newer timestamps is lost or is significantly delayed, the clock expires and triggers the *Actuators*. Therefore, if the server has entered a new cycle but the agent is not aware of it, the *Actuators* will still send the desired actions and the agent will not miss an acting opportunity.

World-Model

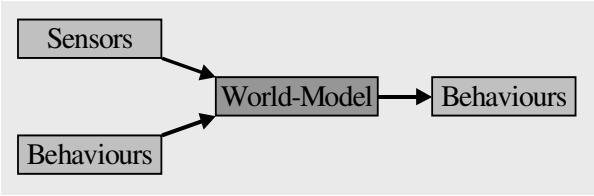


Figure 3.6: World-Model I/O.

The *World-Model* contains a representation of the whole environment, rather than a partial representation like the one provided by visual data from the *Sensors*. The environment (i.e. the soccer pitch) contains 22 agents, 4 lines, 53 flags (used for localisation), 2 goals and a ball. Given the large amount of hidden state in visual sensory data, the agent needs to *remember* past sensations in order to maintain a realistic representation of the environment. In order to do so, each object is assigned a *confidence* value. This value represents the likelihood that the object is in its estimated location. Each time the object is seen, the confidence is set to 1. For every cycle the object is not seen, this confidence decreases. If the value drops below a certain threshold, the object is *forgotten* and it no longer affects the decision making process of the agent.

As seen in figure 3.2, apart from the *Sensors*, the *Behaviours* also provide input to the *World-Model*. These *internal updates* are mostly used as a safety mechanism in case visual data is delayed or lost. For example assume an agent has control of the ball and has just issued a kick command. Until new visual data arrives, the *World-Model* contains information regarding the last seen position of the ball, which is within kicking distance from the agent. Therefore, if visual data is delayed or lost for the following cycle, the agent might attempt to kick the ball again. To prevent this, the position and velocity of the ball can be updated *internally* based on the previous kick command that was sent to

the server. This allows for a more accurate representation of the environment, which does not rely on visual data. Since these internal updates cannot take noise into account, the agent refines its estimates upon receipt of newer inputs from the server.

Parameters

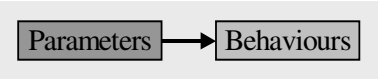


Figure 3.7: Parameters I/O.

In contrast to the *World-Model*, which holds information regarding objects whose values can change over time (e.g. the ball's and opponents' positions), *Parameters* hold information related to attributes that remain *constant* throughout the life-span of the agent. This includes information regarding both the server (e.g. the ball's maximum speed and decay rate, the agent's maximum stamina, etc.) and the agent itself (e.g. kicking and dribbling buffers, passing risk thresholds, etc.). Even though *Parameters* remain constant they directly affect *Behaviours*, since nearly all of the agent's behaviours rely on such values. Currently *Parameters* hold information for over 800 attributes.

3.3 Multi-Threaded Agents

An additional feature of the Essex Wizards MAS, is the multi-threaded implementation of the *sense-think-act* cycle within the agent. This section initially outlines the single-threaded models used so far and gives a brief description of the two most dominant single-threaded implementations used in the RoboCup domain. Following this, a description of the proposed multi-threaded approach is given, together with experimental

results that illustrate its superiority comparing to single-threaded models. The current implementation focuses on POSIX threads, which is the thread application programming interface (API) specified by the standard POSIX 1003.1c-1995. Although POSIX threads are used in this implementation for portability reasons, the higher level design can be implemented in any system with any threads API.

3.3.1 Single-Threaded Agents

According to [Butenhof, 1997] a thread is the set of properties that suggest *continuousness and sequence* within a machine. In other words when a program is executed, a process is created. This process can be thought of as a single *thread* of execution. Of course a process has many additional properties like its own address space, file descriptors and various other data. Up to date, the majority of the implementations in the RoboCup simulation league have been single-threaded (e.g. CMUnited [Stone et al., 1999], ATHumboldt [Burkhard et al., 2000], Andhill [Andou, 1998], RoboLog [Murray et al., 2001], Brainstormers [Riedmiller, et al., 2000]). This essentially means that the initial process generated by the executable file does not create any additional threads. All computations are performed in a serial manner. If a given operation requires the process to be put to sleep (e.g. a read on a blocking socket when no data is available) then the whole execution is suspended.

As mentioned earlier, at a higher level, the agent is responsible for performing three individual tasks. First it needs to *receive* sensory information from the server via a UDP/IP socket. Then the agent has to *think* in order to produce a desired action. Finally this action should be *sent* back to the server via a UDP/IP socket. To perform all these

tasks using a single thread the agent has to use a serial-processing loop like the one shown in figure 3.8.

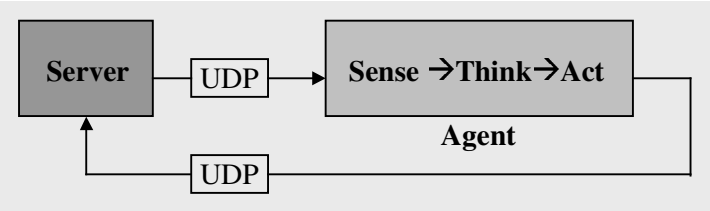


Figure 3.8: Example of a single-threaded agent.

Given that two out of three operations involve I/O on a UDP/IP socket, it is clear that a single-threaded serial implementation puts a significant limitation to the agent's capabilities. Recall that network I/O operations can be extremely slow. Although an agent receives data from the server quite frequently, it cannot afford to wait for data before sending actions. If an agent is put to sleep until new data arrives, it risks missing acting opportunities, which significantly decreases its overall performance. This is the main reason for not using blocking I/O in single-threaded implementations. Instead, the most popular models are I/O multiplexing and signal driven I/O, with the latter being the most dominant in the recent competitions.

I/O Multiplexing Implementations

I/O multiplexing is the capability to instruct the kernel to notify a process if one or more I/O conditions are ready (i.e. input is ready to be read, or the descriptor can receive new output). In the case of the RoboCup domain, the agent needs to check if data is available from the server, or if it is ready to send an action to the server. One of the main uses for multiplexing is when an agent is handling interactive input together with a network socket at the same time. It is worth mentioning that nearly all versions of the server come

together with an interactive sample-client that utilises multiplexing. Following this example, a number of teams based their initial implementations round this model. However, in later years as teams improved, multiplexing was found to be inefficient. This section illustrates some of the problems encountered when using such a model.

In multiplexing implementations, after a connection has been established with the server, the agent enters an infinite loop that is normally called the *main-loop* (or message-loop). Inside the main loop, the *select* system call is used to determine whether data has arrived from the server or an action is ready to be sent. If new data has arrived on the socket, the agent analyses and stores the information. Alternatively if an action is ready, the agent sends this action to the server.

```
while (agent is alive)
 {
 Call select to check which descriptor is ready

 if (read descriptor is ready)
  receive and analyse data from the server

 if (write descriptor is ready)
  send an action to the server

 execute thinking process
}
```

Figure 3.9: Pseudo-code implementation of I/O multiplexing.

This approach has several disadvantages. Firstly, it is difficult to accurately time each phase of the agent's cycle. The agent can only have an estimate of the time required to analyse the sensory data, perform all the thinking, and produce an action. Hence, response times (and therefore timing correctness) cannot be guaranteed for the 100ms-interval set by the server. Secondly, this model allows for only one I/O operation to be executed at any given time. In other words, although *select* can wait on multiple descriptors, the main-loop can only perform one task per iteration. Finally, the *select*

function is quite slow since it requires two system calls in order to receive data. Initially, *select* itself has to be executed to check whether data is available, and then if data exists on the socket the agent needs to call *recvfrom* in order to receive that data. Figure 3.9 presents an example of a pseudo-code implementation using I/O multiplexing.

Signal Driven I/O Implementations

Signal driven I/O is used to instruct the kernel to notify the agent by generating a *signal* when something happens on a descriptor. To implement signal driven I/O two signals are normally used, namely, SIGIO and SIGALRM. One is generated when new data arrives from the server through the UDP/IP socket (SIGIO), the other is generated when an interval timer expires which indicates that an action must be sent (SIGALRM). In other words, one signal is used for *sensing* and one for *acting*.

The ability to suspend the main program's execution and enter a signal handler is what gives signal driven I/O the ability to achieve pseudo-concurrency. Pseudo-concurrency in turn can be used to guarantee fixed response times (within a certain resolution). Normally the SIGIO handler is used to analyse sensory input from the server and the SIGALRM handler is used as a trigger to send actions to the server.

Although this model is the most suitable for single-threaded implementations, there are a number of inherent difficulties in its implementation. Firstly, although it is possible to establish handlers for multiple signals (e.g. SIGIO, SIGALRM, etc.), only one can be executed at any given time. Therefore, if a signal is generated while another handler is being executed, certain precautions have to be taken to avoid signal loss. A common solution is to *mask* additional signals while a signal handler is being executed, and *unmask* them when it has finished.

```
Signal handler for SIGIO
 {
 // SIGALRM has just been masked
 Receive and analyse data from the server
 // On exit, SIGALRM will be unmasked
 }

Signal handler for SIGALRM
 {
 // SIGIO has just been masked
 Send actions to the server
 // On exit, SIGIO will be unmasked
 }

Agent's main loop
 {
 Install SIGIO and SIGALRM handlers
 Set an interval timer to 100ms
 // (100ms is the duration of a cycle)

 while (agent is alive)
  execute thinking process
 }

Operations performed by the kernel
 {
 if (interval timer has expired)
  generate a SIGALRM signal
 if (new data arrives on the socket)
  generate a SIGIO signal
 }
```

Figure 3.10: Pseudo-code implementation of signal driven I/O.

However, this introduces additional problems with SIGIO signals. For example, consider the case where two more datagrams arrive on the UDP/IP socket (i.e. visual and audio data from the server) while a signal handler is already being executed. This will cause the SIGIO signal to be generated twice. Unfortunately, since the signal is masked, when the signal handler returns, the system can only observe one SIGIO signal (given that signals cannot be queued). This will force the agent to read the second datagram but the third one will remain in the queue until another SIGIO is generated. A solution would be to keep polling the UDP/IP socket at regular intervals to ensure that no packets remain in the queue. Clearly this decreases the performance of the agent since it consumes valuable

resources that could be used to perform other computations. An example of a pseudo-code implementation for signal driven I/O is given in figure 3.10.

3.3.2 A Multi-Threaded Approach

Instead of a single-thread, a process can have multiple threads, sharing the same address space and performing different operations independently and without affecting each other. This way the agent can have a separate thread for each of the tasks of the *sense-think-act* cycle. Such agents are called *multi-threaded agents*. The proposed multi-threaded model can be seen in figure 3.11.

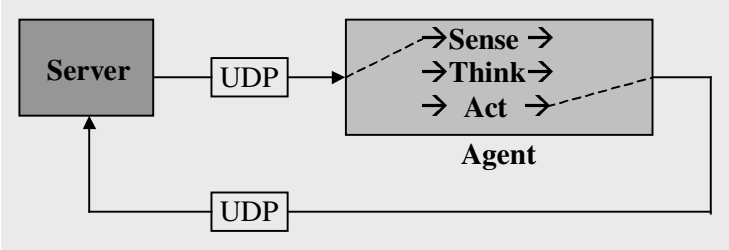


Figure 3.11: Example of a multi-threaded agent.

Inside the agent, *sensing*, *thinking*, and *acting* run concurrently using separate threads as depicted in figure 3.11. The sense-thread is responsible for receiving and analysing sensory information whilst the act-thread is responsible for timing and sending actions. The dashed arrows in figure 3.11 indicate these communication links between the server and the agent. Although this approach is logically similar in terms of information flow to the single threaded agent (shown in figure 3.8), it has significant differences and advantages in terms of computational performance.

More specifically, it can be seen that as the think-thread is independent, it is not involved in slow I/O operations on the UDP/IP sockets. Therefore, although slow I/O

operations affect the whole process in a single-threaded model, the concurrent execution of the multi-threaded scheme minimises these delays. In addition, if for example, the sensor thread fails, the remaining two threads will continue operating as normal. Note that since the information contained in the *World-Model* will quickly become inaccurate, the resulting behaviour will soon resemble that of a blind (and deaf) human footballer.

As with single-threaded implementations, an important first step is to specify the I/O model that is going to be used. This now becomes trivial, since there are separate execution threads for both input and output. For example, the sense-thread can now use a blocking I/O connection. Given that this thread is dedicated to receiving data, there is no need to waste processing time querying the socket as to whether data is available. With a blocking connection, the *recvfrom* call will put the sense-thread to sleep until new data arrives on the socket. However, putting the sense-thread to sleep does not affect the execution of the other two threads that can proceed as normal. When data arrives on the socket the sense-thread will be awakened and it will analyse the next available packet from the server. Also note that this approach does not allow for packets to be lost, or queue up. This would only happen if the server transferred data faster than the thread could analyse it.

Equally simple is the implementation of the act-thread, which is responsible for sending actions to the server at regular intervals. Recall that the current architecture supports a hybrid of event-triggered and time-triggered agents. Therefore, the act-thread can also remain idle for most of the cycle's duration until it is signalled to send actions to the server. As mentioned earlier, the event-trigger used is the *entering to a new cycle*, which is normally detected by checking the timestamp of sensory data received by the server. The time-trigger is implemented using the *gettimeofday* function together with a conditional variable. More specifically, the conditional variable is set to a time slightly

greater than the duration of the cycle. If it expires before the event-trigger has taken place, the act-thread starts sending actions to the server. Whenever the agent enters a new cycle, the conditional variable is reset. This method provides highly accurate timing comparing to the single-threaded approaches described earlier. It also enables the agent to guarantee timing correctness, which is something single-threaded approaches failed to do.

```
SensorThread
 {
 while (agent is alive)
  receive and analyse data from the server
 }

ActuatorThread
 {
 while (agent is alive)
  {
  wait for new cycle event OR
  wait for conditional variable to expire

  send actions to the server
  }
 }

Agent's main loop
 {
 Create SensorThread
 Create ActuatorThread

 while (agent is alive)
  execute thinking process
 }
```

Figure 3.12: Pseudo-code implementation of a multi-threaded approach.

Finally the think-thread is a dedicated thread that decides the best course of action given the current state of the environment. It should be noted that since both the sensing and acting threads remain idle for most of the duration of the cycle, the think-thread is free to consume the majority of the available resources. A pseudo-code implementation of the proposed multi-threaded approach is shown in Figure 3.12.

Multi-threaded programming is quite a large topic. Describing issues such as scheduling policies, how threads compete for resources, or how often pre-emption occurs falls outside the scope of this thesis. A detailed description on multi-threaded programming can be found in [Butenhof, 1997].

3.3.3 Experimental Results

To compare the two models a simple experiment was carried out using a uni-processor PC with an Intel Pentium II 450MHz processor, running RedHat Linux 5.2. The purpose of this experiment was to evaluate only the timing correctness of the models described above, since logical correctness is based on the individual implementations. The test required an agent to connect to the server and send a *turn* action each cycle. In addition, the agent is required to receive and analyse all sensory information to simulate a real game situation. The sensory analysis functions are identical for all implementations.

Table 3.1: Cycle loss.

Cycle loss	Single-threaded I/O Multiplexing	Single-threaded Signal Driven I/O	Multi-threaded Approach
1 agent	1651 (27.52%)	24 (0.4%)	0 (0%)
6 agents	1653 (27.55%)	1020 (17%)	0 (0%)
11 agents	1656 (27.60%)	3294 (54.9%)	4 (0.07%)

Initially, a single agent was tested on the field. The same experiment was then carried out using 6 and 11 agents to examine the efficiency of each model as computational resources decrease. The results for a 6000-cycle-test (a game's duration) are illustrated in table 3.1. The table entries indicate the number of cycles for which an agent failed to send an action, or send more than one action, and hence did not meet the timing

requirements. Adding more agents has a similar effect as extending the thinking process. In both cases, adding more computations depletes system resources. This is to test the scalability of each implementation in terms of number of operations per cycle. As can be seen, I/O multiplexing (using the *select* system call) has over 27% cycle loss even when only 1 agent is used. However, multiplexing is highly scalable since this percentage does not increase noticeably for 6 or even 11 agents. On the other hand signal driven I/O performs significantly better in the single agent trial with only 0.4% cycle loss. Unfortunately it degrades rapidly as more agents are added, with 17% cycle loss for 6 agents and nearly 55% cycle loss for 11 agents.

The results generated by the multi-threaded implementation illustrate that there is nearly 0% cycle loss for all trials. This proves the superiority of multi-threaded approaches in terms of performance and scalability. It should be noticed that multi-threaded implementations are even more efficient when used on multi-processor hardware.

3.3.4 Single versus Multi Threaded Implementations

Comparing the multi-threaded implementation against the single-threaded ones is not an easy task, since both have various advantages and disadvantages. However, certain points about the two approaches can be highlighted given the requirements of the RoboCup and similar domains.

One of the main benefits of multi-threaded implementations is that they provide a model that allows more efficient exploitation of the agent's natural concurrency. Various computations can be performed while waiting for slow I/O operations to complete. This is a significant advantage considering the number of I/O operations per game.

In addition, using multiple-threads provides a modular programming model that clearly expresses relationships between independent events within the agent. Designing a multi-threaded agent illustrates various dependencies and synchronisation requirements between different modules. In addition, mixing CPU-intensive processes with I/O intensive processes results in better utilisation of resources even on uni-processor hardware. Clearly, the performance improves further on multi-processor configurations.

Finally, the underlying principles of multi-threaded implementations are similar to both multi-processing (a thread being considered by many as a lightweight process) and multi-processor implementations (in robotics concurrent behaviours are frequently implemented in parallel hardware) providing additional compatibility between this multi-threaded simulation and a physical implementation of the agent.

On the other hand, there are some disadvantages in multi-threaded programming. Initially, a multi-threaded model is normally more complicated than a single threaded one. Careful design is required to keep track of synchronisation protocols and program invariants. In addition, there are computational overheads in multi-threaded implementations but, in this case, it is argued that the performance benefits accrued in avoiding I/O related delays far outweigh the thread management overheads. Finally, the POSIX standard does not provide an interface for object oriented (OO) languages. The current implementation uses an interface based on work presented in [Lippman and Stroustrup, 1988].

3.4 Design Patterns

Another property of the Essex Wizards MAS is the use of design patterns in the development process. Most hardware engineering disciplines have handbooks that

describe successful solutions to known problems. For instance, an aircraft designer will not design an aeroplane using the laws of physics. Instead, standard successful designs that have been proven useful in the past will be reused. The small increase in performance by starting from scratch normally is not worth the extra cost. In the recent years, due to the increased complexity of software projects, and the tendency towards OO design, such patterns begin to emerge in software engineering disciplines.

Unfortunately, new designers are overwhelmed by the options available to them and tend to fall back on non-OO techniques they have used in the past. This section aims to describe how design patterns have been used in order to improve the development of a key component within the Essex Wizards architecture. Although the focus of this section will be on the m*ediator* pattern, a number of other patterns have been used. For example [Hunter, 2002] uses *prototype*, *factory method*, *singleton*, *decorator*, *template method*, *bridge*, and *chain of responsibility*, in implementing various parts of the agent including the Essex Wizards positioning system [Hunter et al., 2000]. Details for these design patterns can be found in [Gamma et al., 1995] [Coplien, 1992]. A more detailed description of the *mediator* pattern is given primarily because it is an important component of the proposed architecture. In addition, certain extensions have been introduced in order to provide a solution that works in a multi-threaded environment.

3.4.1 Why Design Patterns

As research on artificial intelligence (AI) advances, the complexity of the development of intelligent systems grows rapidly. However, developers are normally faced with similar software design challenges involving robustness, efficiency, and extensibility. Most of these challenges at a higher level are independent of the application specific requirements.

Although certain patterns have been successfully implemented in order to tackle these issues, they are rarely documented. Consequently this knowledge remains hidden in the minds of developers or buried within complex source code [Schmidt, 1997]. Although there are many similarities in the different forms of AI software in general, the focus of this thesis is on MAS software, which has both *inherent* and *accidental* complexities.

Inherent complexities stem from fundamental challenges of developing MAS software. For instance, most MAS are prohibitively complex for handcrafted decision making approaches. In such cases the use of machine learning techniques can greatly improve the decision making process of an agent [Kostiadis and Hu, 1999]. In addition, co-operation between agents is also an important challenge in MAS [Tambe, 1997]. Co-operation further introduces the challenges of communication [Stone and Veloso, 1998] and role variation [Coradeschi and Karlsson, 1998]. Finally, in adversarial domains, an additional challenge is that of opponent modelling [Riley, 1999].

Accidental complexities associated with MAS software come from limitations with conventional tools and techniques. For example, low-level application programming interfaces (APIs) like Sockets are tedious and error prone. Another source of accidental complexity arises from the widespread use of algorithmic design to develop MAS software. Algorithmic design yields non-extensible software architectures that cannot be customised rapidly to meet changing application requirements [Schmidt, 1997]. OO techniques provide principles, methods, and tools that significantly reduce the complexity and cost of developing software [Fayad and Schmidt, 1997]. The main benefits of OO arise from the emphasis on modularity, reusability and extensibility. A design pattern represents a recurring solution to a software development problem within a particular context [Gamma et al., 1995]. Patterns capture abstract designs and software

architectures in a systematic format that can be readily comprehended by developers [Schmidt, 1997].

3.4.2 Designing a Multi-Agent System

An agent is any system that can be viewed as perceiving its environment through sensors and acting upon that environment through effectors [Russell and Norvig, 1995]. A multi-agent system is a collection (or team) of such agents. In order to create a coherent team of agents, the entire design process and the agent architecture must be built with the team in mind. "Collaboration must be designed into systems from the start; it cannot be patched on [Grosz, 1996]." Also during the design process the issues of robustness, reusability, and extensibility must be considered. It has been shown [Gamma et al., 1995] that design patterns can help the potential developer deal with a number of problems including decomposing a system into objects, determining their granularity, and specifying their interfaces and implementation.

In addition to the complexity of building each module within an agent, there are normally multiple people involved in the implementation. Under these circumstances, integration and testing becomes a difficult task. The proposed design should be modular, highly flexible and scalable. All the modules in the Essex Wizards architecture described above, are represented by various objects, in a fully OO design. Due to the tight coupling of the independent components within the agent, the design should allow easy modification, addition and removal of various modules, and provide a flexible interface for the inter-communication between them. Finding an acceptable solution to this problem is not a trivial task.

3.4.3 Multi-Threaded Mediator Pattern

The well-known m*ediator* design pattern [Gamma et al., 1995] has been used to tackle this problem. The idea behind the mediator pattern is to define a module that encapsulates how sets of modules interact. Mediator promotes loose coupling and modules within the agent can communicate without explicitly referring to each other. Given the multi-threaded nature of the Essex Wizards team, an extension to the standard mediator pattern is required. Shared data, like the ones contained in the *World-Model*, need to be protected from accidental reads/writes and various communication links need to be synchronised. This section briefly describes the mediator pattern and presents the extensions added in order to build a design pattern that can operate in a multi-threaded environment.

Breaking an agent into individual modules encourages the distribution of behaviour among these modules. However, such distribution can result in a structure with many inter-connections. Although partitioning a system normally enhances reusability, having too many inter-connections tend to reduce it again. Numerous inter-connections make it more difficult for a component to work without the support of others. In addition, making significant changes to the overall behaviour of the system becomes unnecessarily difficult, since behaviour is distributed among many modules.

These problems can be avoided by encapsulating collective behaviour in a separate mediator object as shown in figure 3.13. The mediator controls and co-ordinates the interactions of the various modules within the agent. The mediator acts as a communication hub between components, keeping them from referring to each other, and hence reducing the number of inter-connections between them. Each module is only required to know its mediator and all communication with other modules is done

indirectly through this channel. An additional benefit comes from the localisation of the overall behaviour. The system can easily change its behaviour by modifying or replacing just the mediator.

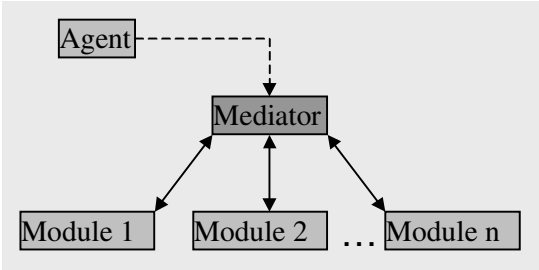


Figure 3.13: Mediator pattern.

Although the mediator design pattern promotes loose coupling and reduces inter-connections between modules, it also tends to centralise the overall control of the agent. In other words there is a trade-off between complexity of module interaction and complexity in the mediator. Since one object is used to encapsulate communication between all others, the mediator can become more complex than any individual component within the agent. The potential developer must be careful in order to prevent the mediator itself from becoming monolithic and hard to maintain.

As mentioned earlier, the Essex Wizards MAS is based on a multi-threaded implementation. The traditional mediator pattern is not capable of behaving as intended in such an environment. Assume for instance that the *Sensors* have just received new information and started updating the *World-Model*. While the update is taking place, the *Behaviours*, which operate from a different thread, might try and read the contents of the *World-Model*. Since the update has not yet been completed, the *World-Model* contains up-to-date information for certain parts of the environment, and older information for the rest of the environment. In most cases this will result in sub-optimal or even incorrect

actions being taken. Therefore, a synchronisation protocol needs to be integrated into the current mediator pattern.

The most common and general way to synchronise data access between threads is to ensure that all read/write operations on shared data are mutually exclusive. The POSIX threads API that has been used in the Essex Wizards implementation provides mutual exclusion using a special form of Edsger Dijkstra's semaphore, called a mutex [Butenhof, 1997]. Mutexes allow a thread to lock specific data items so that no other thread can operate on them at the same time. In the example mentioned above the *Sensors* would have to lock the contents of the *World-Model* while the update was taking place. This would result to the *Behaviours* being put to a halt until the update was finished and the *Sensors* unlocked the specified mutex. A schematic representation of this example can be seen in figure 3.14.

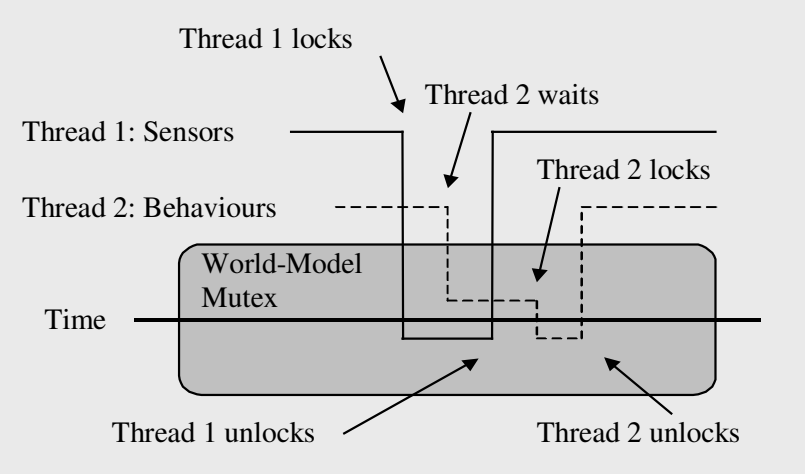


Figure 3.14: World-Model mutex operation.

To enable the mediator pattern to function correctly in a multi-threaded environment, similar synchronisation had to be integrated into the various communication channels between the mediator and specific modules. There is always a trade-off between successful synchronisation protocols and real-time responsiveness. Since various threads

can be halted, it is important to lock mutexes for the minimum time required for a read or write operation to complete, and this must occur only when necessary. In the current implementation, synchronisation takes place mainly within the mediator, just before control is passed to a module. This way, operations within modules become *atomic*, and synchronisation is *transparent* to the individual components of the agent. Unfortunately, there are cases where the mediator is not involved in a given operation, yet synchronisation between shared data needs to be applied. These cases are rare and occur when certain operations need to take place within a single module without the need for communication with other modules. In such cases the mediator is not directly involved and synchronisation can only take place within the given module.

3.5 Discussion

The RoboCup domain provides with a complex simulated environment, which allows for the full utilisation of the benefits of OO techniques and design patterns. The proposed design and architecture have been successfully implemented into the Essex Wizards MAS. Unfortunately, there are no performance measures for design methodologies or agent architectures. It is therefore quite hard to argue in favour or argue against a particular approach.

One way to obtain an *indication* regarding the overall performance of a team, is by competing with various other teams that use different approaches. By observing certain features during each game (e.g. possession time, attacking opportunities, etc.), a useful evaluation for a given system can be obtained. Recall that the aim of the RoboCup domain is to provide a common testbed where a wide range of technologies can be integrated and examined. For that purpose there are numerous RoboCup competitions

every year. The main event is an annual international competition with a number of participating teams from universities around the world. Given that competitions are not controlled experiments, their results are not presented as scientific validation of the proposed approach.

The first competition the Essex Wizards participated in was the third international RoboCup competition (an event with 36 teams competing in the simulation league), held in Stockholm, during 1999. The Wizards qualified as the first team in their round robin group and went on to achieve a combined score of 59-9 over their opponents, finishing in third place. The following year, during the first European RoboCup championship (an event with 14 teams competing in the simulation league), the Wizards qualified as the second team in their round robin group and went on to achieve a combined score of 53-4 over their opponents, finishing in third place. Detailed team descriptions can be found in [Hu et al., 2000] [Hu et al., 2001] [Hu et al., 2002].

3.6 Related Work

This chapter introduced the functional requirements specification (FRS) for agents wishing to operate in the RoboCup and similar domains. A description of the architecture was given, including the components that make up the agent and the ways these components work together. In addition, the multi-threaded framework for real-time agents was presented. Finally, the multi-threaded mediator pattern, a key component within the agent, was described together with the benefits of applying design patterns for the development of MAS.

The real-time requirements presented in section 2.1 are based on [Nissanke, 1997]. The description of the Unix I/O models (section 2.2) is based on [Stevens, 1998]. [Burns

and Wellings, 1997] provide useful material for the development of real-time systems. [Butenhof, 1997] illustrates the benefits of multi-threaded applications and describes how such applications can be developed using the POSIX API. A high-level approach to distributed systems and client/server application design can be found in [Coulouris et al., 1995]. A lower-level approach, including network programming using UDP/IP and TCP/IP (Transmission Control Protocol/Internet Protocol) sockets, can be found in [Stevens, 1998]. Software design patterns were originally introduced by [Gamma et al., 1995] who present a variety of examples including creational, structural, and behavioural patterns.

Over the years a large number of agents architectures have been proposed, addressing different features an agent should have and targeting different domain requirements [Wooldridge and Jennings, 1995a] [Wooldridge and Jennings, 1995b]. Several taxonomies have also been introduced such as [Nwana, 1996] and [Franklin and Graesser, 1997]. One of the most recent taxonomies, [Muller, 1998], classifies agent architectures as reactive, deliberative, interactive, or layered. (There is also a separate category for architectures that do not fit any of these profiles). Purely reactive architectures, such as [Brooks, 1986], are mainly designed to achieve robust behaviour, rather than optimal behaviour. Agents make decisions based on very limited information regarding their environment, normally using simple situation-action rules. This simplicity of reactive architectures makes them well suited for real-time applications [Agre and Chapman, 1990]. On the other hand, purely deliberative architectures [Laird et al., 1987] [Rao and Georgeff, 1995] [Georgeff and Ingrand, 1989] maintain an internal representation of the environment, which directly affects the decision-making process of the agent. The third category, interactive agents, deal with co-ordination and co-operation among distributed intelligent agents [Normal et al., 1997] [Burmeister and Sundermeyer, 1992]. Finally,

layered architectures [Muller, 1996] [Ferguson, 1992] structure the functions of an agent into hierarchically organised layers that interact with each other in order to achieve coherent behaviour of the agent as a whole.

In recent years, two major areas of computer science have started converging. *AI* methods are moving towards realistic domains requiring real-time responses, and *real-time systems* are moving towards more complex applications requiring intelligent behaviour [Musliner et al., 1995]. This convergence introduced the field of *real-time AI*. The main challenges of real-time AI are highlighted in [Musliner et al., 1995] whereas [Garvey and Lesser, 1993] present a survey of deliberative real-time AI research. In order to develop agents that are capable of both intelligent behaviour and real-time responses, the chosen architecture must balance the need for deliberation and the need for reactivity. [Jung et al., 1997] and [Guessoum and Dojat, 1996] argued that to achieve robust real-time performance a purely reactive agent architecture would be more suitable. However, the tasks that can be achieved using this approach are limited because of the lack of any cognition. On the other hand a purely deliberative approach may be unable to provide the real-time responses required by real-world applications. This is in agreement with [Bussmann and Demazeau, 1994] who have shown that neither purely reactive nor purely deliberative architectures are capable of meeting the requirements imposed by complex real life environments. In fact, a number of applications, both in hardware [Occello and Demazeau, 1994] and software [Fasciano, 1995] domains, rely on hybrid architectures in order to cope with both the complexity, and the real-time constraints of their environments. However, there are a number of ways one can construct hybrid architectures. One approach is to have two or more different subsystems (e.g. one that is highly reactive and one that is highly deliberative) and combine their responses. Examples of such systems include [Howe et al., 1990], [Hayes-Roth, 1990], and

[Musliner et al., 1993]. More integrated approaches, like [Zilberstein, 1993] and [Horvitz and Rutledge, 1991], provide different responses depending on the time criticality of the task. Normally more time for deliberation results in outputs of better quality.

Although many features from different architectures in the literature have inspired this research, any architectural design should be driven by the requirements of the environment the agent is expected to operate in. There is always a trade-off between keeping the architecture generic enough for use in domains with similar characteristics [Vidal and Buhler, 2001] [Wooldridge et al., 2000] [Horling, 1998] and optimising for the domain at hand [Stone and McAllester, 2001] [Stone and Veloso, 1999a]. Given the requirements of the RoboCup domain, agents need to combine real-time *reactive-like performance* together with *deliberative-like intelligent decision-making* in a *co-operative interactive-like fashion*. To that extend, the Essex Wizards architecture realises a reactive *sense-think-act* cycle together with a *world-model* (a component seen in deliberative architectures) that holds detailed information about the environment. More importantly, the *sense-think-act* cycle is implemented in a concurrent way (or parallel if multi-processor hardware is available), using POSIX threads, in order to ensure bounded response times. To the best of the author's knowledge, there are no existing multi-threaded agent implementations in the RoboCup domain. Note that there is a difference between multi-threaded MAS like [Menczer and Monge, 1999], which implement each agent using a single-thread, and multi-threaded agents. Also, there is a difference between a multi-threaded agent and a multi-threaded programming language for MAS such as Gaea [Nakashima et al., 1995], which was used to build the GAMMA [Noda, 1998] team. The co-operative aspects of the Essex Wizards MAS are thoroughly discussed in the following chapters of this thesis.

Chapter 4

Kanerva Based Generalisation (KaBaGe)

4.1 Overview

In simple learning tasks, estimates of value functions can be represented using a table with one entry for each state or for each state-action pair. Although this approach is very clear and easy to use, it is limited to tasks with small numbers of states and actions. Most problems normally have continuous or large discrete state spaces; some have large or continuous action spaces. Using tables to store estimates of value functions in such cases makes inefficient use of experience and normally means impractical memory requirements. An additional problem is that of the time and data needed to fill such tables accurately. Surely, there must be some more compact representation than a table.

The problem of learning in large spaces is tackled through generalisation techniques, which allow compact representation of learned information and transfer of knowledge between similar states and actions. In large smooth state spaces, it is expected that similar states will have similar values and similar optimal actions. Therefore it should be possible to use experience gathered through interaction with a limited subset of the state space and produce a good approximation over a much larger subset. This kind of generalisation is often called function approximation because it takes examples from a

desired function and generalises in order to produce an approximation of the entire function.

In most reinforcement learning (RL) architectures, the storage of a variety of mappings is required, including State → Action (policy), State → Reward (value functions), and (State × Action × State)→[0, 1] (transition probabilities). Some of these mappings can be learned using straightforward supervised learning. Popular function approximation techniques for supervised learning include neural network methods [Rumelhart and McClelland, 1986], fuzzy logic [Berenji, 1991] [Lee, 1991], and cerebellum model articulation controller (CMAC) [Albus, 1981] based methods. Other mappings, especially the policy mapping, need specialised algorithms because training sets of input-output pairs are not available. Numerous generalisation techniques, both over input and over actions, are presented in [Kaelbling et al., 1996]. In this chapter the interest is on a special case of gradient-descent methods known as linear methods. In linear methods, the approximate function is a linear function of the parameter vector. Linear methods can be very efficient in practice both in terms of data and computation [Sutton and Barto, 1998]. Whether or not this is true depends critically on how the states are represented in terms of features. Some of the most popular feature representation techniques include coarse coding [Hinton, 1984], tile coding [Albus, 1981], and radial basis functions (RBF) [Broomhead and Lowe, 1988]. In this chapter the focus is on a feature representation method based on work originally presented in [Kanerva, 1988] also known as Kanerva-coding.

The terms *tile coding* and *Kanerva-coding* are due to [Sutton and Barto, 1998]. Although a number of systems have combined linear methods and reinforcement learning in RoboCup [Stone and Sutton, 2001] [Stone et al., 2001] and other domains [Waltz and

Fu, 1965] [Sutton, 1996], to the best of the author's knowledge there are no reinforcement learning systems that use Kanerva-coding.

4.2 Why Kanerva-Coding

On tasks with very high dimensionality, methods such as tile coding and RBF networks become impractical. This is due to the exponential increase in computational complexity being proportional to the number of dimensions. However, the main idea of representing states by a list of features and then mapping those features linearly to an approximation should work even on large tasks. The only requirement is to keep the number of features from scaling explosively. To achieve this, the complexity of the target function should be kept separate and distinct from the size and dimensionality of the state space. One might have a 500-dimentional task, where only one of the dimensions happens to matter. In other words, the real source of the problem is the complexity of the target function and not the dimensionality of the state space. In fact, adding dimensions to the task should be almost without consequence if the complexity of the needed approximations remains the same. Unfortunately, methods such as tile coding and RBF networks do not work this way since complexity increases exponentially with dimensionality even when the complexity of the target function remains the same.

On the other hand, Kanerva-coding, based on [Kanerva, 1988], does not suffer from such problems. This is due to the complexity being unaffected by dimensionality and being only limited by the complexity of what is approximated. The trick in Kanerva-coding is to choose binary features that correspond to particular prototype states. These prototypes are selected from the entire state space. The strength of Kanerva-coding is that

the complexity of the functions that can be learned depends on the number of features and has no relationship to the dimensionality of the task.

4.3 A Brief Introduction to Kanerva's Memory Model

This section is based on work presented in [Kanerva, 1988] [Anderson, 1995] and [Gallant, 1993]. Kanerva in his model of Sparse Distributed Memory (SDM) tackles the issues of:

- How *records* in memory should be organised so that they can be retrieved in the right way under the right circumstances.
- How to construct a physical memory that allows for the proper storage and retrieval of a *record* from neuron-like components.

A *record* is defined as a sequence of memory items that represents the experiences of an organism. Although the agent's world-model presented earlier in this thesis is not based on Kanerva's SDM model, the KaBaGe-RL decision-making module uses Kanerva-coding as a generalisation method. Hence it is necessary to give a brief outline of the theory behind Kanerva's SDM model and explain how it works.

4.3.1 Neurons as Address Decoders

One of the main elements of Kanerva's SDM is the address decoding neuron (ADN). ADNs belong to a certain class of neurons that have their input coefficients (and their thresholds) fixed over the entire life of an organism. Given this interpretation, the n-tuple

of input coefficients determines an n-bit address. This address represents a pattern to which the neuron responds most readily, with the threshold of the neuron controlling the size of the region of similar addresses to which the neuron responds. Although the input coefficients of a given neuron are assumed to be constant over time, different neurons can have different input coefficients.

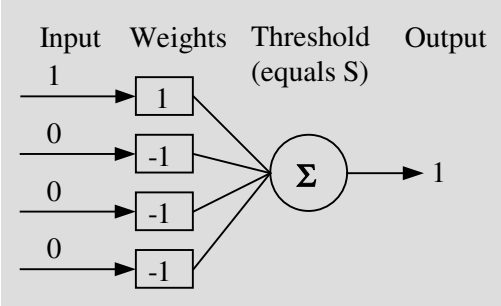


Figure 4.1: Address of a neuron.

The address, *a*, of a neuron with input coefficients *w*, where $w = \{w_0,...,w_{n-1}\}$, is defined as the n-bit input pattern that maximises the weighted sum of the given neuron. This occurs when the inhibitory inputs are zeros and the excitatory inputs are ones (assume for now all weights are either 1 or –1). For example for n = 4 and weights $w_0 = 1$, $w_1 = -1$, $w_2 = -1$, $w_3 = -1$, such an n-bit input pattern (or address) would be 1000. The *maximum weighted sum*, *S(w)*, is defined as the sum of the positive coefficients:

$$S(w) = \sum_{i=0}^{n-1} w_i \quad ,(w_i > 0) \tag{4.1}$$

The point opposite the neuron's address, *a'*, yields the *minimum weighted sum*, *s(w)*, which is the sum of the negative coefficients:

$$s(w) = \sum_{i=0}^{n-1} w_i \quad ,(w_i < 0) \tag{4.2}$$

For brevity, let S stand for the maximum weighted sum $S(w)$ and s stand for the minimum weighted sum $s(w)$. In the example above, $S = 1$ and the pattern corresponding to it is 1000 (figure 4.1), and $s = -3$ and the pattern corresponding to it is 0111.

Perhaps the most important element of the ADN is its threshold. This is because the value of the threshold directly affects the response region of the neuron. When the threshold c is in the range $s < c \leq S$, the output of the neuron is 0 for some input patterns (or addresses) and 1 for others. If the threshold is above S, the output will always be 0. If it is below s, the output will always be 1. By a proper choice of threshold ($c = S$), a neuron can respond to just one address, acting like an address decoder. If the neuron's address is thought of as a *prototype* (or target pattern) and an address x is thought of as a sample pattern, the neuron recognises x, if x is sufficiently close to (or within distance $S - c$ of) the neuron's address. It is important to note that the comparison is indifferent to the location of the mismatched bits. Therefore to accept only closely matching patterns, c must be set to a value almost as high as the maximum weighted sum S. As the threshold is lowered, more and more patterns are recognised.

4.3.2 Sparse Memory

Consider the problem of storing information in a computer memory. Some data in the form of long binary words are to be stored in such a way so that they can be retrieved at a later stage when needed. The simplest way to accomplish this task is to store each word in a convenient location and make a note of where it was stored in a list. In other words, for each word we need a storage location and an address decoder. When that particular

word needs to be retrieved, its address decoder is used to find the location into which it was stored. The word can then be retrieved, and used as needed. Although this approach is very simple and makes no assumptions regarding the contents of the memory, it places a heavy burden on the indexing mechanism.

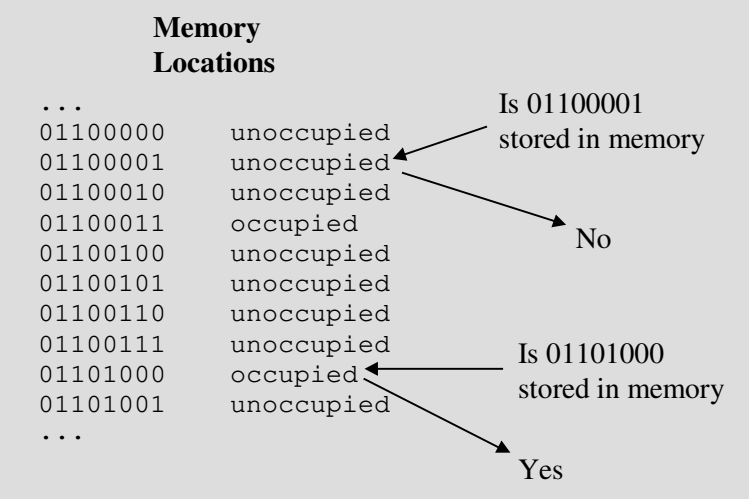


Figure 4.2: Sparse memory.

An alternative is to use the data stored in memory as addresses to the memory. Assume that the data set to be stored consists of 100-bit words. That means 2^{100} different possible binary vectors could be stored. Suppose the memory has 2^{100} storage locations. Then, the contents of the word to be stored can be used as their address in memory. This is also known as Kanerva's *unifying principle*. To locate a word in memory, one only needs to check whether the specified location in occupied (figure 4.2). Although this method uses a one-step lookup to locate a word, it introduces unrealistic memory requirements. The vastness of the address space N (where $N = 2^n$) becomes a significant problem. Even with a modest n = 100, the model requires 2^{100} storage locations. However there is no need for 2^{100} storage locations considering that a century has less than 2^{32} seconds. In addition, if the memory had 2^{100} storage locations, most of them

would be unoccupied. Therefore, there must be a better way to store and retrieve a given data set.

The method proposed by Kanerva has the following characteristics:

- The storage locations and their addresses are given from the start. Therefore, only the contents of the locations are modifiable.
- The storage locations are very few in comparison with 2^n (the memory is sparse).
- The storage locations are distributed randomly in the $\{0,1\}^n$ address space.

These predefined locations Kanerva calls *hard locations*. Instead of 2^{100} storage locations, the memory can consist of say 2^{20} hard locations that are randomly distributed across all the 2^{100} possible words. Since the available locations are less than the possible words to be stored, an improved storage mechanism is needed.

4.3.3 Distributed Storage

Suppose a given word is to be stored in memory. One way would be to find the nearest hard location to the value of the word and store the word there. If the distribution of locations is uniform, a location close to the contents of the word should be available. Given such a method for storing each word, retrieval would be carried out in a similar manner. The best match to a target word would be the word stored to the nearest occupied hard location. However, simply looking at some radius around the target word for a match is not very accurate. To improve the performance of the memory model, the storage procedure can be modified.

This leads to another main concept in Kanerva's SDM, namely *distributed storage*. The idea of distributed storage is that many storage locations participate in a single read or write operation. In other words, instead of storing a single example of the word, several examples are stored by putting copies of the input data into all the hard locations within some radius of the word to be stored. Conventional computer memories have only one location active at any one time. Distributed storage gives the memory the appearance of a random-access memory with a very large address space and with data retrieved on the basis of *similarity*. In other words, if the word η is stored at the address ξ, then reading from ξ retrieves η. However, what is more important, reading from an address x that is sufficiently similar to ξ retrieves a word y that is even more similar to η than x is to ξ. (The similarities are comparable since the address to the memory and the data are elements of the same metric space N). The mathematical feasibility of distributed storage has been demonstrated in [Kanerva, 1998].

Unfortunately, this method is likely to store information from several data words on top of each other, creating interference among different words. Kanerva suggests several ways to cope with this problem. The most common one assumes that each hard location does not store a single word, but instead forms the *sum* of all the words it stored during training. Assume an input word is an exact match to a stored word. Suppose that the radius to be checked for matches contains 1000 hard locations. When storing the word, a copy of the input was distributed among those 1000 locations. Also, other unrelated words from other stored data will be stored at some of these locations. However, the most common word will be the stored one since the radius of storage and the radius of retrieval are the same. Therefore, all the 1000 activated hard locations will contain a copy of the stored word. If the data from all 1000 locations is summed up, the result will contain 1000 copies of the desired data and fewer copies of other data that had a different

radius of storage. Therefore, the stored target word will dominate the sum simply because of the statistics of the items used to form the sum. On the other hand, assume the word to be checked and the stored word are not identical but they are similar. In this case, the number of examples of the stored word within the radius of the retrieval will still be large because the two circles (storage and retrieval) will be nearer for these two words, than for other more distant ones.

A schematic representation of distributed storage can be seen in figure 4.3. During storage of the word η at the address ξ, a number of locations near ξ have been used. Retrieving data from the address x causes a number of storage locations to be activated within a certain region from x. All the locations add their contents to form the sum y. Many of the locations will have the data η stored in them because the regions overlap. Therefore, a good approximation of the originally stored data can be retrieved from the sum.

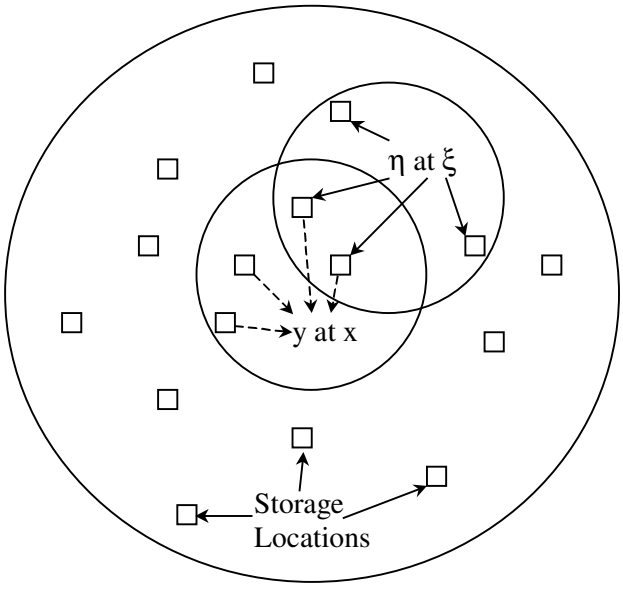


Figure 4.3: Distributed storage.

4.3.4 Storage and Retrieval of Sequences: Prediction

Kanerva also proposed and analysed a simple iterative technique for constructing the location of the best match to a given word, as long as the input and the best match were not too far apart. Assume that the average data word generated by the summing process from the input word is used as the input word for another retrieval step. If the target word and the input word are not too far apart, the resulting average data word will move closer to the target word. Repeated iterations will eventually converge to the nearest location of the best match.

An mentioned earlier, an important question in any memory model is how information should be stored in memory so that it can be retrieved later when the situation warrants it. This requires that:

- The *present* situation is recognised as being *similar* to some situation in the *past*.
- The *consequences* of that past situation can be retrieved.

However, as mentioned earlier, one of the main ideas in Kanerva's SDM is that of the *unifying principle*. In other words, contents of the memory can be used to address memory since the data stored in memory are addresses to the memory itself. This allows a sequence of words to be stored as a pointer chain, and therefore allows the memory to support sequential accessing. Reading from memory repeatedly can then retrieve any word.

4.3.5 Memory for Patterns and Pattern Sequences

The SDM model presented by Kanerva works with long vectors of bits. These vectors can be thought of as patterns of binary features (generalisation to multi-valued features is also feasible [Rao and Ballard, 1995a]). This provides with a memory that can be addressed by large patterns of features and can also store these very same patterns.

Since a pattern can be used both as an address and as a datum, a sequence of patterns can be stored as a pointer chain. The first pattern in the sequence is used as the address in storing the second pattern, the second is used as the address in storing the third, and so on. Any pattern in the sequence can then serve as a starting point for the retrieval of the rest of the sequence.

An important benefit of using such a memory model is that addressing the memory need not be exact. A previously stored pattern can be retrieved with addresses similar to it even if the original storage address is unknown. Therefore reading within the *critical distance* of such an address retrieves a pattern that is closer to the written pattern (on average) than the read address is to the write address. Critical distance is the distance beyond which divergence is more likely than convergence. (Recall that a sequence of read words converges only if the initial address is sufficiently close to the target).

This attracting property is fundamental to sequence recall. For example, to use the memory for recognising a set of patterns, each pattern is stored with the pattern itself as the address. On the other hand, to use the memory for recalling sequences of patterns, each sequence is stored as a pointer chain. In either case the pattern just read is used as the next read address. In addition, since write addresses attract, the initial read-address need not be exact. If it is within the critical distance of some previous write address, (say 3-6 iterations) this will usually suffice to read patterns exactly as written.

The memory groups patterns automatically, allowing two kinds of generalisation:

- To read from memory, the exact address patterns that were used in writing in to memory need not be known.
- If many similar addresses have been used as write addresses, then the individual patterns written with those addresses cannot be recovered exactly. This generalisation is in terms of the features that make up the patterns. In other words, features that are common to most of the patterns in the neighbourhood will stand out as an encoding for a cluster of patterns.

The power of this memory model is based on its ability to retrieve sequences and to generalise. If an organism's past is represented as a sequence of patterns and if this sequence has been stored in memory, surely the pattern representing the present moment can be used as an address to retrieve the consequences of similar moments in the past.

4.4 Implementing the Generalisation Module

Having briefly outlined the theory behind Kanerva's memory model, it is time to demonstrate how this theory can be used to construct a generalisation module for a learning system. Recall that the SDM model is responsible for both storing and retrieving sequences. However, one of the main requirements for KaBaGe is reusability. In other words, the aim is to construct a module that can be added to a number of learning systems without major modifications to the original system's architecture. A direct implementation based on Kanerva's theory would require significant changes in the way the agent perceives and stores information regarding its environment. Since perception is one of the main components on any learning system, producing something that requires

changes in perception and storage of information would seriously compromise reusability. Therefore, KaBaGe does not deal with how information is stored in the system's memory. Instead, it maps the agents' memory onto a binary feature vector and uses randomly constructed prototypes to achieve feature extraction on any input signal, assuming such a mapping is feasible.

4.4.1 Design and Implementation of KaBaGe

KaBaGe is implemented using 4 components connected in a serial manner, as shown in figure 4.4. The output of each component in the series becomes the input to the next. The module is initialised with a number of randomly constructed prototypes. As will be seen shortly, the number of prototypes significantly influences the behaviour of the system. In addition, the way the prototypes are constructed directly affects the performance of the generalisation. Before describing how the prototypes are constructed and how many should be used, a description of the components within KaBaGe is required.

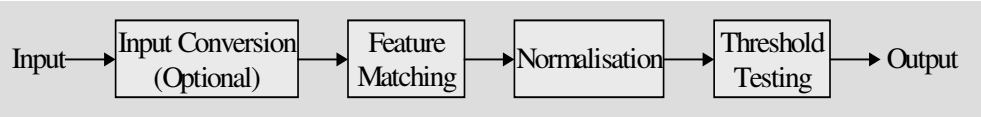


Figure 4.4: The KaBaGe module.

The raw input signal initially passes through a conversion process. Recall that Kanerva-coding works with binary feature vectors (although generalisation to multi-valued features is feasible [Rao and Ballard, 1995a]). Note that this step is optional, since the original input signal might already be in a binary format. If there is no acceptable

way to perform this conversion, the multi-valued version of Kanerva-coding should be used instead. Throughout this thesis, all input signals are either binary, or they are converted to a binary format. As will be shown, KaBaGe works even on complex environments with continuous input variables. The binary input signal is then passed to the second component within KaBaGe, the feature-matching component. This component is responsible for matching the input signal against each prototype. The result of this bit-wise comparison between the two vectors (i.e. the input and the prototype) is an integer greater than or equal to zero that indicates the number of bits at which they match. These values are then normalised by dividing with the number of dimensions of the input vector. Finally, the normalised values are compared against a fixed threshold. If the normalised value is greater than or equal to the threshold the output (for this prototype) is 1, otherwise it is 0. A result of 1 indicates that the given feature exists in the input, whereas a result of 0 indicates that it does not.

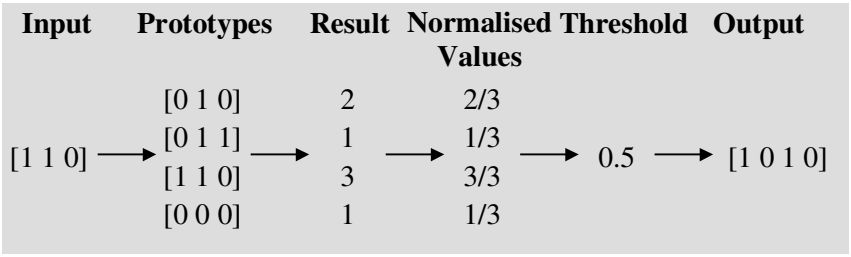


Figure 4.5: Generalisation example.

This process is best illustrated with the following example (figure 4.5). The input vector [1 1 0] is presented to the KaBaGe module. Since it is already in a binary format the first step (i.e. that of input conversion) is omitted. In this example the system has 4 prototypes. The input vector is matched against each prototype to produce a matching result. For example the first prototype [0 1 0] has only 2 bits in common with the input vector (the second and third). Hence, the matching result is equal to 2. The columns

Prototypes and *Result* (in figure 4.5) correspond to the *Feature Matching* component of the KaBaGe module (figure 4.4). The result of each match is then divided by the size of the input vector to produce a normalised value between 0 and 1. Finally the normalised values are compared against the threshold to decide whether a given feature exists in the input or not. In this example prototypes 1 and 3 produce a matching value greater than or equal to the threshold whereas prototypes 2 and 4 do not. Hence the output is the vector [1 0 1 0].

Something worth mentioning is that although the original input vector was of size 3, the output vector is of size 4. In other words the result is dependent on the number of prototypes and not the dimensionality of the input vector. By selecting an appropriate number of prototypes, KaBaGe can produce an output vector smaller, equal, or larger in dimensions than the original input vector.

4.4.2 Prototype Construction

As mentioned in the previous section, the way the prototypes are constructed directly affects the performance of the generalisation. In addition, the example showed how the dimensionality of the output vector is dependent on the number of prototypes and bares no relationship to the input vector. Therefore it is important to find acceptable ways for both selecting the number of prototypes and defining their construction method.

According to [Sutton and Whitehead, 1993], some of the most effective and popular methods for on-line learning include RBF networks [Moody and Darken, 1989], CMAC networks [Albus, 1981], and self-organising maps [Kohonen, 1990]. All these methods are strictly incremental and can handle non-stationarity. In addition, they all share a similar two-layer structure. The first layer *expands* the original input representation into a

high dimensional feature space, and the second layer maps that expanded representation to the final answer. This general architecture is known as learning with an *expanded representation* (ER).

Following results by [Gallant and Smith, 1987] and [Prager and Fallside, 1988], [Sutton and Whitehead, 1993] have shown how ER systems in which the expanded representation is constructed with a significant random component can perform very well. They call such systems *random representation* (RR) systems. Their results indicate that RR systems can outperform nearest neighbour and perceptron learning methods without requiring unreasonably large representations. In addition they showed how RR systems, with little increase in representation size, could handle moderately high dimensional input spaces including many irrelevant inputs. Finally their results demonstrate that RR methods show much faster initial learning compared to that of backpropagation networks. Following these encouraging results, KaBaGe is constructed with randomly generated prototypes, and with the number of prototypes within KaBaGe sufficiently larger in number than the dimensions of the original input representation.

A final problem is that of a choice of threshold. An important issue when considering the value of the threshold is that of the density of activation, in other words, the number of prototypes that are active at one time. If the threshold is too low, many of the prototypes will be active for most input vectors. On the other hand, if the threshold is too high, very few prototypes will be active at any one time. If all prototypes are active or inactive at the same times, then they are redundant. The objective is to keep the density of activation within a certain percentage of the total number of prototypes. The exact range for that percentage is dependent on the task at hand, and the total number of prototypes in the system.

4.5 Experimental Results

Before integrating KaBaGe with the rest of the decision-making process, individual testing (also known as unit testing) is required in order to ensure that it is suitable as a generalisation module. For this purpose, a simple classification task together with a simplified learning rule is used. This classification task is the one used by [Sutton and Whitehead, 1993] when testing the performance of RR systems. By using identical testing conditions, the performance of KaBaGe can be compared to that of RR systems, perceptron learning and nearest neighbour methods. Following is the description of the task, which is based on the description presented in [Sutton and Whitehead, 1993].

The implementation considers tasks that are generated randomly as follows. Each input vector consists of 16 randomly generated bits. From these 16 bits, 8 are designated as relevant and 8 as irrelevant. In addition, 1 of the irrelevant bits is designated as the target bit. For each task, 8 prototypical inputs are also generated. Half have a target bit of 1 and the other half a target bit of 0. The *actual target* for a randomly generated 16-bit input is computed by comparing the input to each of the prototypical inputs. The prototypical input with the most matching relevant bits is selected and its target bit used as the *actual target* of the input. Ties are broken according to an arbitrary ordering of the prototypical inputs. Noise is introduced by flipping the *actual target bit* on a random 5% of the examples (in order to keep the testing conditions identical to the ones presented by [Sutton and Whitehead, 1993]).

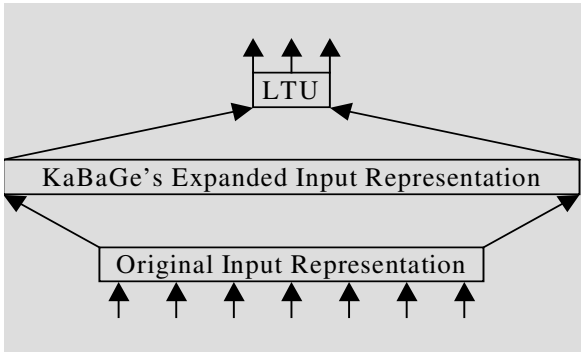


Figure 4.6: General 3-layer architecture using KaBaGe.

The *actual target* for a given input is compared against the *guessed target* that the system produces. The system is a basic 3-layer network. The first layer represents the input, the second layer is the KaBaGe generated feature vector and the third and final layer is a single linear threshold unit (LTU) that determines whether the *guessed target* (i.e. the output) is 0 or 1 (figure 4.6). The output of the LTU is given by:

$$f = \begin{cases} 1, \ if \sum_{j=0}^{m} w_j y_j \geq 0 \\ 0, \ otherwise \end{cases} \tag{4.3}$$

where w are the weights connecting each prototype to the final LTU. All the weights are initially set to 0 and then adjusted according the a normalised, cross-entropy perceptron rule:

$$\Delta w_j = \frac{a}{N}(f^* - p) y_j \tag{4.4}$$

where $\alpha > 0$ is a learning rate, N is the number of prototypes that were active, f^* is the target value for the current example, and p is the estimated probability that $f^* = 1$ given by:

$$p = \frac{1}{1 + e^{-\sum_{j=0}^{m} w_j y_j}} \tag{4.5}$$

This learning rule is the one used in [Sutton and Whitehead, 1993], and is a version of the perceptron suited to binary classification problems [Hinton, 1989]. The least-mean-square (LMS) or normalised LMS rule would be more appropriate, if the overall problem were one of minimising squared error.

The network was run with random representations of size 100 (RR100) and 5000 (RR5000). It was also run using KaBaGe with 100 (KaBaGe100) and 5000 (KaBaGe5000) prototypes. Each configuration was tested for 200, 1000, 3000, and 5000 example runs. The number of correct guesses given by each configuration was averaged over 20 repetitions of the whole procedure. The results can be seen in figure 4.7. This figure shows how the performance of the system improves as the representation size increases.

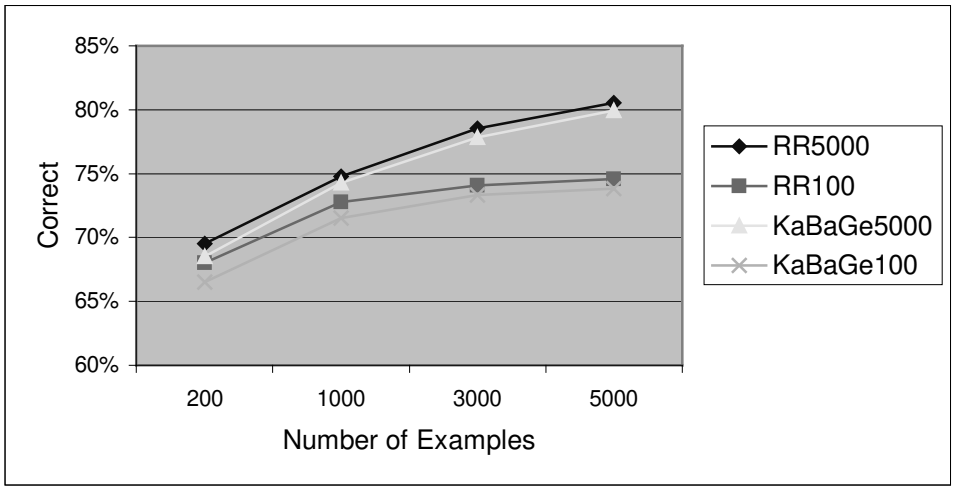


Figure 4.7: Learning curves on first task.

Figure 4.8 shows the total number of correct guesses as a function of the representation size. Trials were run for 5000 examples with representation size values of 20, 100, 200, 1000, 3000 and 5000 for both RR and KaBaGe systems. Results were averaged over 20 repetitions of the whole procedure. Again it can be seen that the performance of the system only improves as the representation size increases. These results are in agreement with the ones presented by [Sutton and Whitehead, 1993]. This was expected, since KaBaGe is an instance of a RR system.

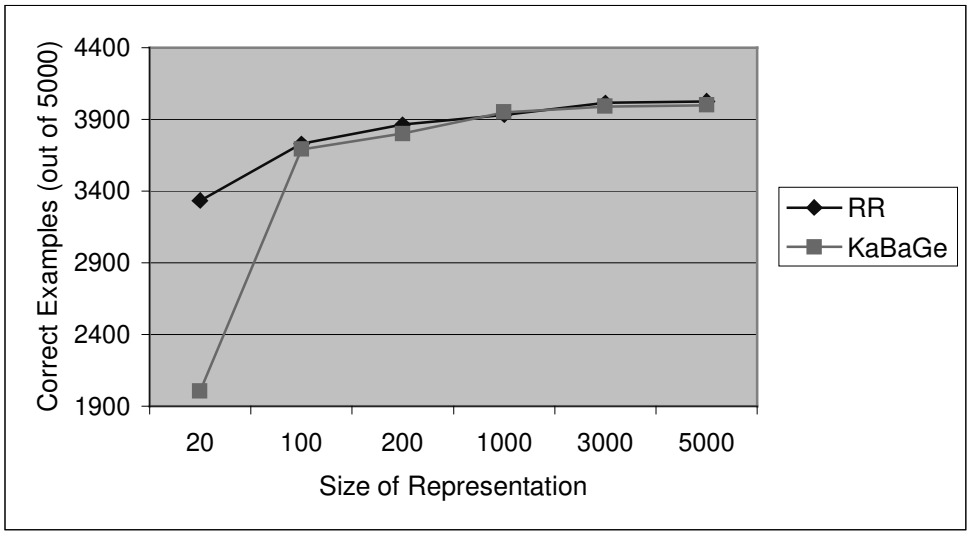


Figure 4.8: Performance versus representation size on the first task.

It should be noted that [Sutton and Whitehead, 1993] have shown the performance of the RR5000 to be significantly better comparing to nearest neighbour and perceptron learning methods. Also note that [Sutton and Whitehead, 1993] used the first weight as bias with the corresponding input signal defined always to be 1.

4.6 Related Work

This chapter demonstrated how Kanerva's SDM model could be used as a generalisation component for tasks that involve large state spaces. This was a necessary first step towards the development of the KaBaGe-RL decision-making module, which is presented in chapter 6. To the best of the author's knowledge, there have been no attempts to combine Kanerva's SDM together with RL methods.

Since function approximation in general is a large research topic, this section considers generalisation techniques that can be used with RL. For such methods [Bertsekas and Tsitsiklis, 1996] [Boyan et al., 1995] and [Kaelbling et al., 1996] present some useful material. The benefits of linear methods and the advantages of Kanerva-coding (section 4.2) are highlighted in [Sutton and Barto, 1998]. Section 4.3 is based on the description of the SDM model of [Kanerva, 1988] [Anderson, 1995] and [Gallant, 1993]. Figures 4.2 and 4.3 are based on similar figures from [Anderson, 1995]. Figure 4.6 is based on a similar figure in [Sutton and Whitehead, 1993]. Finally, the learning task used in section 4.5 was originally presented in [Sutton and Whitehead, 1993].

An important problem in connectionist networks is that of catastrophic forgetting. This occurs when old information is destroyed by newly learned data. According to [French, 1991], a SDM can easily store new information without interfering with previously stored information as long as the representations do not significantly overlap. If the memory becomes saturated, there is interference among representations, and learning new information begins to interfere with old. However, as long as the memory is designed so that saturation is not likely to occur, such a system can handle storing new data without forgetting older data.

The advantages of SDM models have made their use possible in a number of applications ranging from vision systems to mobile robot navigation. For instance [Rao and Ballard, 1995b] proposed a scheme for expression-invariant face recognition that employs a fixed set of natural basis functions to generate multi-scale iconic representations of human faces. Object recognition has been a central subject in the field of computer vision. Recognition of human faces is a challenging sub-problem of this field. Apart from the difficulties introduced due to the changing viewing conditions, computational methods also have to deal with faces being complex non-rigid stimuli that defy easy geometric characterisations and form a dense cluster in the multidimensional space of input images. [Rao and Ballard, 1995b] represent a face using a collection of iconic feature vectors that are automatically extracted from specific locations in the input image. Following this, a topographically organised SDM is used to learn the association between the appearance of a face as given by its feature vectors and the identity of the face.

[Rao and Ballard, 1995a] described a general purpose object indexing technique that combines the virtues of principal component analysis (PCA) with the favourable matching properties of high dimensional spaces to achieve high precision recognition. The indexing mechanism uses an active vision system in conjunction with a modified form of a SDM which facilitates interpolation between views and provides a convenient platform for learning the association between an object's appearance and its identity. More specifically, object indexing is performed using multi-scale iconic feature vectors as components of a SDM. Experimental results indicate that the proposed indexing scheme is remarkably tolerant to moderate changes in viewing conditions caused by occlusions, illumination changes, scale changes and rotations in 3D. There are 3 main modifications for the SDM in Rao's design. Initially, the addresses are no longer binary but correspond

to multi-valued response vectors whose range is determined by the range of filter outputs. In addition, the normalised dot product is used as the distance metric, instead of the Hamming distance. Finally, the prototypes are not picked at random but are picked according to the distribution of the input data [Keeler, 1988]. This final modification can provide a significant increase in performance for certain distributions of data. In certain cases, if prototypes are picked randomly, a large number of locations is never activated while other locations are selected so often that their contents resemble noise.

[Rao and Fuentes, 1996] presented a general framework for the acquisition of perception-based navigational behaviours in autonomous mobile robots. A possible method for memory-based navigation is to use an associative memory in the form of a lookup table that associates a large set of perception vectors from different locations with a corresponding set of actions required to navigate to a desired location. However this approach has a number of drawbacks. Initially, the address space formed by the perception vectors is usually quite large and therefore, storing fixed reference vectors for every possible scenario becomes infeasible. In addition, the training time increases drastically as the lookup table size increases. Finally, lookup tables relying on nearest-neighbour methods normally fail to generate the appropriate responses to novel situations. [Rao and Fuentes, 1996] have tackled these problems by using a SDM, which is modified to use competitive learning in order to adapt its sensorimotor space and radial interpolation to compute motor output vectors. The main modification was that the memory accepts multi-valued perception vectors from an arbitrary distribution and stores an associated multi-valued motor vector in a distributed manner in the data space.

[Turk and Gorz, 1995] reported an implementation of a SDM for the Connection Machine [Hillis, 1985] and its application. Their implementation consists of three major parts, namely, the address module, the contents module and external registers. The SDM

has been implemented in a native object-oriented programming style, namely, the message passing style. An SDM object is realised as a closure, containing all the necessary methods and variables, such that the co-existence of any number of SDMs in the front-end memory can be managed. A similar attempt to develop simulators for SDMs is reported in [Rogers, 1989].

Over the years, a number of modifications to Kanerva's original SDM model have been proposed [Jaeckel, 1989a] [Jaeckel, 1989b]. Recall that when accessing a SDM a number of locations that are *close to* (according to Hamming distance) the input address are activated. Following Jaeckel's work, [Karlsson, 1995] introduce an efficient mechanism for finding these locations. Their implementation is suitable for execution on sequential machines, it is much faster, and gives approximately the same signal-to-noise ratio.

[Kristoferson, 1995] considered a SDM with randomly chosen hard locations, in which an unknown number T of random data vectors have been stored. They then propose a method to estimate T from the content of the memory with high accuracy.

[Zboril, 1997] introduced the idea of using an SDM either as an auto-associative memory or as a hetero-associative one. An interesting application demonstrated in [Zboril, 1997] is the combination of the two SDMs where an initial input pattern is presented to the hetero-associative SDM through a multiplexer. The output is then presented to the auto-associative SDM, which is shown to produce a more precise result.

Within the RoboCup domain, work on generalisation is presented in [Stone and Veloso, 1999b]. The generalisation component of the Team-Partitioned Opaque-Transition Reinforcement Learning (TPOT-RL) module uses action-dependent features rather than discretising the various dimensions of the state space. Each possible action is evaluated locally based on the current state of the world using a fixed function. This

function does not produce the expected long-term reward of taking the action. Instead it classifies the likely short-term effects of it. In addition, in the multi-agent scenario, other than one output of the fixed function for each action, the feature space also involves one coarse component that partitions the state space among the agents. This generalisation process reduces the complexity of the learning task by creating a small feature space that partitions the state space into regions. Each agent needs to learn how to act only within its own partition.

Chapter 5

Learning to Act Using Reinforcement Learning

5.1 Overview

The complexity of most multi-agent systems prohibits a hand-coded approach to decision making. In addition, the huge state space, large action space, and real-time requirements of the RoboCup domain make decision making an even more challenging task. To tackle the problem of decision making in RoboCup and other similar domains, the use of machine learning (ML) methods becomes necessary. ML is concerned with the question of how to construct computer programs that can automatically improve their performance with experience. The field of ML is very large and provides various methods for solving a number of different tasks [Mitchell, 1997]. However, the focus of this thesis is on reinforcement learning (RL), and more specifically Q-learning [Watkins, 1989].

RL addresses the question of how an agent that can sense and act in its environment can learn to choose optimal actions in order to achieve its goals. A significant advantage of RL methods is that the agent learns by trial and error without being told which actions to take. Instead, after each action the agent receives a reward that gives an indication of how well it performed. The agent must therefore try different actions and discover the ones that produce higher rewards.

This chapter explains why RL was the preferred method for action selection together with a brief explanation of how RL methods work. In addition, a basic co-operative task is used to demonstrate the applicability together with the strengths and weaknesses of RL methods in the RoboCup and similar domains. Parts of this chapter have originally appeared in [Kostiadis and Hu, 1999] and [Hu et al., 1999].

5.2 Why Reinforcement Learning

The main advantage of RL methods is that they provide a way of programming agents by reward and punishment without needing to specify how a task is to be achieved. Each agent is assumed to have a set of sensors that provide with some indication of the state of the environment and a set of actuators that allow it to act and change the state of the environment. After each action, the agent receives a reward that indicates how well it performed. The learner is not told which actions to perform but instead must discover the ones that produce higher rewards by trial-and-error. In most cases, actions taken in a given state directly affect subsequent states. Therefore, an action not only affects the immediate reward, but also rewards that will arrive from future states. These two characteristics, trial-and-error searching and delayed rewards, are the two most important features of RL.

Perhaps the most widely studied methods in ML are those of supervised learning. Supervised learning methods normally require a set of examples provided by a knowledgeable external supervisor. Although this is an important kind of learning, alone is not adequate to produce autonomous systems that can learn by interaction. It is normally impractical to collect a set of examples of desired behaviour that are both correct and can represent all the situations which an agent might face. Learning is most

beneficial when an agent is required to act based on its existing experience when presented with an unknown situation. This is particularly the case in complex environments where there are large numbers of states and possible actions.

Moreover, learning methods that require supervision by a human expert are not always applicable. In complex domains like RoboCup, human experts might not always be in agreement as to which is the best action given a state. For example it is not always clear whether in a particular situation an agent should favour passing over shooting or vice versa. In fact, in the game of backgammon, Tesauro's TD-Gammon [Tesauro, 1995] has changed the way human experts play from certain positions. In opening play, most human experts now prefer *splitting* rather than *slotting* after the success this technique had with TD-Gammon. It is worth mentioning that *slotting* was the preferred choice by human players for the past 30 years. This is quite a significant result since TD-Gammon received no training from human backgammon experts. TD-Gammon required very little backgammon knowledge, yet learned to play extremely well after learning the game by competing against itself.

5.3 A Brief Introduction to Reinforcement Learning

This section is based on work presented in [Mitchell, 1997] [Kaelbling et al., 1996] [Sutton and Barto, 1998]. The standard RL model considers agents that are connected to their environment via perception and action as shown in figure 5.1.

On each step of interaction the agent receives an input which provides some indication of the state s of the environment. The agent then chooses an action a to execute. As a result, the environment rewards the agent (normally with a scalar reward $r \in R$) and moves

to a next state. An intuitive way to understand the agent's interaction with the environment is:

$$s_0 \xrightarrow{\alpha_0}{}_{r_0} s_1 \xrightarrow{\alpha_1}{}_{r_1} s_2 \xrightarrow{\alpha_2}{}_{r_2} \cdots \tag{5.1}$$

where $\{s_0, s_1, s_2, \ldots, s_n\}$ is the set of states, and $\{\alpha_0, \alpha_1, \alpha_2, \ldots, \alpha_n\}$ is the set of actions. The real values r_t indicate the immediate reward an agent receives when performing action α_t from state s_t. It is apparent that an action taken by the agent not only affects the immediate reward, but also rewards that will arrive from future states. For example the total reward received by the agent after the interaction described above is:

$$r_0 + \gamma r_1 + \gamma^2 r_2 + \ldots + \gamma^N r_N = \sum_{t=0}^{N} \gamma^t r_t \tag{5.2}$$

where $\gamma \in [0, 1)$ is a constant (also known as the discount factor) that determines the relative value of delayed versus immediate rewards, and N is the expected lifetime of the agent. Note that N could be unknown or ∞. The agent's task is to learn a policy, (a mapping from states to actions) $\pi: S \rightarrow A$, which maximises the expected sum of these rewards, with future rewards discounted exponentially.

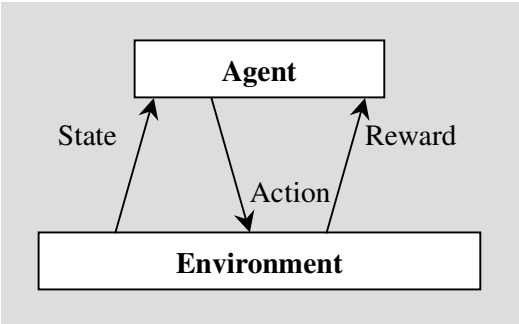


Figure 5.1: Standard RL model.

5.3.1 Models of Optimal Behaviour

Before learning a policy that chooses optimal actions, a model of optimality has to be defined. As surveyed in [Kaelbling et al., 1996], three models have been the subject of the majority of work in this area:

Finite Horizon Model

This model works under the assumption that at any given moment in time, an agent should optimise its expected reward for the next h steps, and is defined as follows:

$$M_F = E(\sum_{t=0}^{h} r_t) \tag{5.3}$$

where r_t represents the scalar reward received t steps into the future. The agent need not worry about what will happen after that. This model has two variations. In the first, the agent follows a non-stationary policy, i.e. one that changes over time. On the first step the agent will chose an *h-step optimal action*, on the second step an *(h-1)-step optimal action* and so on. Finally in the last step the agent will choose a *1-step optimal action* and terminate. The second variation is known as *receding-horizon control*. In other words, the agent always takes an *h-step optimal action*. The finite horizon model is not always suitable since the precise length of the agent's life might not be known in advance.

Infinite Horizon Discounted Model

This model takes the long-run reward of the agent into account, but rewards that are received in the future are geometrically discounted according to a discount factor $\gamma \in [0, 1)$ as described in equation (5.2) above. It can be expressed as follows:

$$M_{\infty} = E(\sum_{t=0}^{\infty} \gamma^t r_t) \tag{5.4}$$

This model is conceptually similar to that of receding-horizon control described above. However, the discounted model is more mathematically tractable, which is the main reason for the wide attention it has received. The experiments described in this and the following chapter use an infinite horizon discounted model.

Average Reward Model

In this model the agent is supposed to take actions that optimise its long-run average reward, defined as follows:

$$M_A = \lim_{h \to \infty} E(\frac{1}{h} \sum_{t=0}^{h} r_t) \tag{5.5}$$

Such a policy can be seen as the limiting case of the infinite horizon discounted model as γ approaches to 1. The disadvantage of such a model is that it is unable to distinguish between a policy that gains a large amount of reward in the initial phases and the other that gains a large amount of reward in the final phases of the agent's life.

5.3.2 Difficulties in RL

Although RL offers a wide range of advantages, unfortunately a reinforcement learner also has a few problems to overcome. This section provides a brief description of the main obstacles that are encountered and suggests possible ways to overcome them.

Delayed Rewards

As described earlier, an agent's action determines not only the immediate reward, but also (at least probabilistically) the next state of the environment. Hence the agent must take into account not only the immediate reward, but also the reward it can receive from the future states that the current action will lead. The models of optimality described above decide how the future should be taken into account. The agent however must be able to learn an optimal policy based on rewards that can take place arbitrarily far in the future. Determining which of the actions in its sequence are to be credited with producing the eventual rewards is also known as the *temporal credit assignment* problem.

Delayed rewards are considered to be one of the most important problems in RL. The longer the reward sequences are, the more trials the learning algorithm requires and thus the longer it takes to converge. It will be shown later on that since RL is a back-propagating method, it is not very efficient for long-term goals. However, it can be very powerful from states that are near the goal state(s).

By introducing intermediate goals through distribution of responsibilities, each agent will only need to achieve a *local* goal that is dependent on its role. For example in the RoboCup domain, a goalkeeper need not worry about how to score a goal, which is a task numerous stages away from its normal responsibilities. The goalkeeper's task is to make a successful pass to a nearby defender. By introducing various roles, and assigning different goal states to each role, the responsibilities are distributed between team members. Although in RoboCup the ultimate goal is to score against the opposition, this is not, and should not be treated as the aim of every individual agent.

Exploration versus Exploitation

Another important problem is that of exploration versus exploitation. The agent is responsible for the actions it chooses and these actions influence the distribution of training examples. This raises the question of which experimentation strategy produces most effective learning. At any given point an agent faces a trade-off in choosing whether to favour exploration or exploitation. Exploiting a known action guarantees a certain reward, however exploring new actions might result in higher rewards for a given state. What the agent chooses to do heavily depends on how long it is expected to live for. The longer it lives, the worse the consequences of prematurely converging on a sub-optimal action. A number of ways to tackle this problem are described in [Kaelbling et al., 1996].

For the experiments in this chapter, a greedy strategy is used. Greedy strategies choose the action with the highest estimated payoff. Since the task presented here is a basic one, a greedy approach provides the most suitable solution. However, the experiments in the next chapter use a greedy strategy combined with optimistic action initialisation. This ensures that all actions will be tried at least once, because any that are untried appear better (since they were optimistically initialised) than those that have been tried in the past and have been accurately valued. Therefore the agent is forced to explore all actions in its action space. When all actions have been tried at least once, the agent chooses the one that appears to be optimal.

Partially Observable States

In most real-world problems and also in the RoboCup domain, an agent's sensors cannot perceive the entire state of the environment at each time step. The sensors normally

provide only partial information regarding the agent's surroundings. In such cases, one might have to combine previous information together with new sensory data when choosing an action. Also in some occasions the best policy might be one that specifically chooses actions that improve the observability of the environment. There are a number of policies that can help an agent resolve the partial observability of a state. They are mainly categorised in state-free deterministic policies, state-free stochastic policies, and policies with internal state.

State-free deterministic policies deal with the problem of partial observability by ignoring it. In other words, the agent tries to find an optimal policy with partial information regarding its environment. On the other hand, state-free stochastic policies map observations to probability distributions over actions. [Jaakkola et al., 1995] developed an algorithm for finding *locally-optimal* stochastic policies. Unfortunately, the computational expense of finding a *globally-optimal* policy makes this approach impractical.

The most suitable approach for RoboCup and similar domains is that of policies with internal state. This is mainly because such policies rely on the fact that the only way for an agent to behave effectively is to use memory of previous actions and observations to disambiguate the current state. This is why the agent is equipped with a world-model as described in chapter 3. The agent's world-model holds a probabilistic representation of the entire environment based on both the current and previous sensations. Each object in the environment is assigned a probability of existence for its present location. When the object is seen this probability is set to 1 and then decreases as time progresses if the object is not seen. A forgetting threshold is used to determine whether a given object will affect the action selection process.

5.3.3 RL Methods

[Sutton and Barto, 1998] described 3 fundamental classes of methods for solving RL problems, namely, dynamic programming (DP), Monte Carlo methods, and temporal difference (TD) learning. Each method has its own strengths and weaknesses. Certain differences in their requirements, properties, efficiency, and speed of convergence are used to determine the best method given a problem. A brief description follows, which explains the reasons for choosing Q-learning, a TD algorithm, as the most suitable method for problems similar to the ones presented in this thesis.

Dynamic Programming

DP refers to a collection of algorithms that can be used to compute optimal policies given a perfect model of the environment as a Markov decision process (MDP) [Bellman, 1957] [Bertsekas, 1995]. Although DP methods are important from a theoretical perspective, they are of limited utility in RL both because of their perfect world-model requirements and because of their computational expense.

Monte Carlo Methods

Unlike DP, Monte Carlo methods do not assume complete knowledge of the environment. They only require experience, i.e. sample sequences of states, actions, and rewards from interactions with the environment. Monte Carlo methods solve RL problems by averaging sample returns. To ensure well-defined returns, only episodic tasks are considered. In other words, to use Monte Carlo methods, experience should be divisible into episodes and all episodes must eventually terminate no matter what actions are

selected. Value estimates and policies are changed only upon completion of an episode. Therefore, Monte Carlo methods can be considered incremental in an episode-by-episode sense but not in a step-by-step sense. The term Monte Carlo is often used for any estimation method involving a significant random component. In the context of RL, [Sutton and Barto, 1998] use the term for methods based on averaging complete returns.

Temporal Difference Learning

TD learning is a combination of Monte Carlo and DP ideas. Like Monte Carlo, TD methods can learn directly from experience without requiring a model of the environment's dynamics. Like DP, TD methods do not require for an episode to end before updates can be made. Instead, they update estimates based in part on other learned estimates. In the RoboCup and similar domains, the complexity of the environment rules out the use of DP methods. Also the fully incremental property of TD provides a significant advantage over Monte Carlo methods since an episode could take a long time to end. For example in the RoboCup domain, the end of an episode could be when a team scores a goal. This however might only happen once or twice in a whole game. Delaying all learning until an episode's end might be too slow. Common TD algorithms include Sarsa [Rummery and Niranjan, 1994] (the term Sarsa was actually introduced by [Sutton, 1996]), Q-learning [Watkins, 1989], and Actor-Critic methods [Barto et al., 1983] [Sutton, 1984].

5.4 Implementing a Q-Learning Based Decision Making Module

At each discrete time step t, the agent senses the current state s_t, and chooses an action α_t to perform. The environment responds by rewarding the agent $r_t = r(s_t, \alpha_t)$, and producing the succeeding state $s_{t+1} = \delta(s_t, \alpha_t)$. Note, that the functions r and δ are not known to the agent.

The agent's task is to learn a policy $\pi: S \rightarrow A$, for selecting actions based on the current state. The agent should look for a policy that will maximise the discounted cumulative reward achieved from an initial state s. In other words, the agent's learning task is to find a policy π, which will maximise the discounted cumulative reward for all states. Such a policy is called an optimal policy, and is denoted by π^*.

Unfortunately, it is difficult for an agent to learn π^* directly given the available training data. Recall that the agent is not provided with examples of the form state-action (s, α). The only training information is a sequence of immediate rewards. Given this kind of training information it would be much easier to learn a numerical evaluation function defined over states and actions, and then implement the optimal policy in terms of this evaluation function [Mitchell, 1997].

Such an evaluation function is $Q(s, \alpha)$, which is defined so that its value is the maximum discounted cumulative reward that can be achieved starting from state s and performing action α. Q-learning was developed by [Watkins, 1989] as a method that can acquire optimal policies when the reward and transition functions are unknown to the agent. The main advantage of Q-learning lies in the fact that the value of Q for each state-action pair, summarises in a single number all the information needed by the agent in order to determine the discounted cumulative reward that will be gained in the future.

This means that an optimal action can be chosen at each state without ever conducting a look-ahead search to determine what the precise results from that action are going to be. Moreover the agent is now able to select optimal actions even if it has no knowledge of the reward and state transition functions r and δ. A convergence proof for Q-learning can be found in [Watkins and Dayan, 1992].

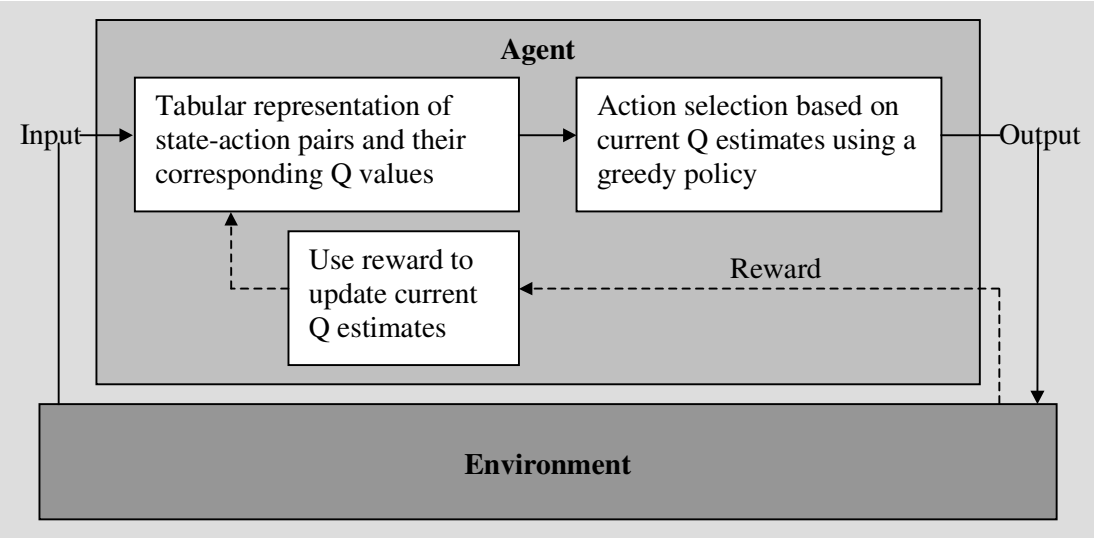


Figure 5.2: The agent's learning process.

For the experiments presented in this chapter, the simplest form of Q-learning has been used, i.e. tabular one-step Q-learning. According to this method the agent maintains a table with all possible state-action pairs *(s, a)* and refines the values of this table after each iteration using the following update rule:

$$Q(s_t, a_t) \leftarrow r + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) \tag{5.6}$$

The whole learning process can be seen in figure 5.2. After receiving the input signal, the agent determines the state of the environment. For that given state the lookup table contains a set of all possible actions, together with their Q value estimates. The action with the highest payoff (i.e. highest Q value) is selected and executed by the agent. The

reward from the environment is then used to update the table that holds the Q value estimates.

5.5 The 2 Attackers versus 1 Defender Challenge

The performance of a Q-learning agent is tested using the *2 Attackers versus 1 Defender* challenge (or 2v1 for short), a subtask of the RoboCup domain. The 2v1 challenge was originally introduced by [Noda et al., 1996] and is the simplest form of co-operative behaviour in the RoboCup domain, involving 2 teammates against 1 opponent. The aim is to test whether the agents can *learn* that in a number of situations *co-operative behaviour* is more beneficial. Once an agent is equipped with the necessary low-level skills (e.g. ball interception and kicking), the next step is to learn higher-level co-operative behaviours such as passing. This basic case also aims to illustrate that reinforcement learning can be used to *learn* an optimal policy in a domain such as RoboCup. Recall that sensing and acting is noisy, and the environment does not provide with a reinforcement signal after an action is executed. Instead, the agent has to perform *observational self-reinforcement* i.e. reward itself based on subsequent (noisy) sensory inputs. For the experiments presented in this chapter the reward is either +1 if a goal is scored, or –1 if the defender intercepted the ball. Note that self-reinforcement has previously been studied within the RoboCup domain only by [Stone and Veloso, 1999b]. Within the RL domain in general, [Mataric, 1994] has used progress estimators as intermediate rewards in order to help robots reach a goal location.

In the 2v1 challenge there is a team of 2 *Attackers* trying to score a goal against a *Defender*. The whole scenario takes place within a limited region in front of the defender's goalmouth. Each episode starts with the attackers in possession of the ball.

As soon as the defender gains possession of the ball the episode ends and the agents are reset for another episode (having possession of the ball means that the ball is within kicking distance from the agent). The 2v1 challenge can be fully represented using 4 states and 2 actions as shown in figure 5.3.

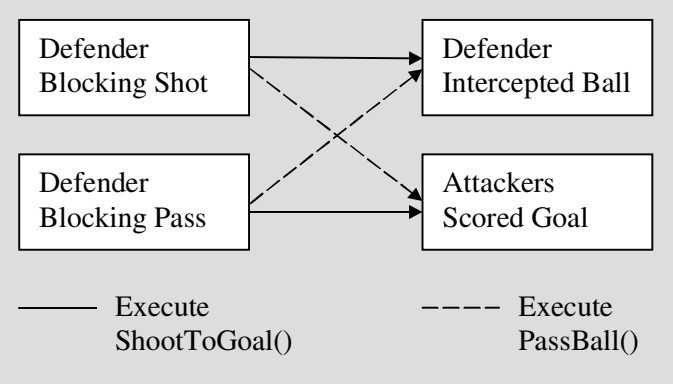


Figure 5.3: State graph of the 2v1 challenge.

The 2 states shown in figure 5.4 represent the 2 initial positions of the 2v1 scenario. Agent number 1 is the defender, agent number 2 is the attacker with the ball (the *Q-learner*), and agent 3 is the second attacker. In state 1, the defender is placed in a position from where it can block a direct shot, but cannot intercept a pass between the 2 attackers or a shot from attacker 2 (agent number 3). In state 2, the defender is placed in a position from where it can block a pass, but cannot intercept a direct shot. At the beginning of each episode the attacker in possession of the ball (agent number 2) has to make the decision to either:

- **ShootToGoal()** – Kick the ball using maximum power towards the goal in a direction that has the lowest probability of being intercepted by the defender, or
- **PassBall()** – Pass the ball to the second attacker (agent number 3).

The second attacker's action space consists of only 1 action, **ShootToGoal()**. Given the nature of the situation, the offside rule was disabled for these experiments.

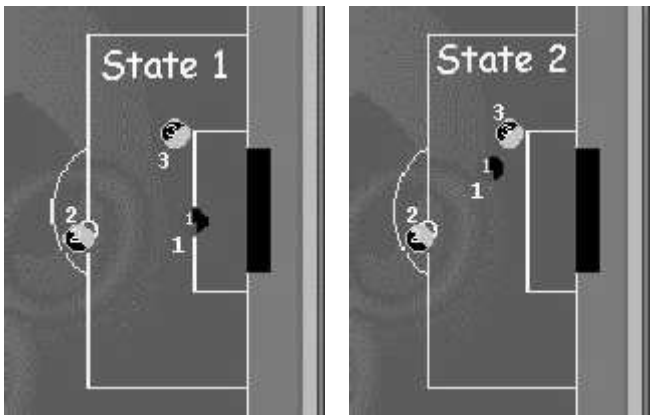


Figure 5.4: Initial states for the 2v1 challenge.

The 2v1 challenge is a subtask of robotic football. Since there are fewer agents involved, and the game takes place in a smaller region, the agents can concentrate on the same high-level goal without the need to balance offensive and defensive considerations. Nevertheless, learning to pass in a goal-scoring scenario like the 2v1 is of great importance in full-scale 11 versus 11 games. In fact, given the 2-dimentional nature of the environment, there are very few cases where a goal can be scored from a 1 on 1 situation. In competitions the vast majority of the goals are scored in a similar manner like the one presented here since *lobbing* (i.e. kicking the ball above an agent) is not an option in 2D environments.

5.6 Experimental Results

The performance of the Q-learning attacker was measured by comparing it to that of a random attacker. The random attacker has the same action space as the Q-learning attacker but chooses which action to execute (i.e. **ShootToGoal()** or **PassBall()**) at

random. In other words, the position of the defender and therefore the state of the environment do not affect the decision making process of a random attacker. The results of 150 trials are illustrated in table 5.1.

Horizontally the table indicates the number of successful, failed, and total number of trials. Vertically the table shows the number of passing, shooting, and total attempts. For example entry 2-2 indicates that the agent has successfully chosen to pass the ball, i.e. when the defender was blocking a direct shot, 41 out of 79 times a pass was executed (i.e. 51.90% of the passes were successful). Similarly, the entry below indicates the number of times the agent has incorrectly chosen to pass the ball, i.e. when the defender was blocking the pass.

Table 5.1: Results for a random attacker.

Random Attacker	**PassBall()**	**ShootToGoal()**	Total
Success	41 (**51.90%**)	35 (**49.30%**)	76 **(50.67%)**
Failure	38 (**48.10%**)	36 (**50.70%**)	74 (**49.33%**)
Total	79	71	150

Following the random attacker, a Q-learning attacker was used, which tries to *learn* the optimal action depending on the state it is currently in. The results of 150 trials are shown in table 5.2.

Table 5.2: Results for a Q-learning attacker.

Q-learning Attacker	**PassBall()**	**ShootToGoal()**	Total
Success	61 (**100.00%**)	88 (**98.88%**)	149 **(99.33%)**
Failure	0 (**0.00%**)	1 (**1.12%**)	1 (**0.67%**)
Total	61	89	150

The reason for the extremely high success rate in the decisions made by the Q-learning attacker lies in the back-propagating nature of Q-learning. Initially the learner's Q estimates for all state-action pairs are set to zero. Since both actions have the same Q value (i.e. zero) the agent randomly chooses an action to execute. If it is the wrong action, the agent receives a negative reward (or punishment), and if it is the correct action it receives a positive reward. The updates of the Q estimates for this state after the reward is received force the agent to choose the correct action for all subsequent trials. In other words, once the goal-state has been reached, the learning propagates backwards to the previous states. Since the state space for this basic case is so small, the optimal action is found after just one visit to each initial state. It is worth mentioning that for exactly the same task, experiments carried out by [Noda et al., 1996] using a back-propagating neural network yield a success rate of 68%. Also [Noda et al., 1998] using a C4.5 decision tree yield a success rate of 63.7%.

5.7 Discussion

A number of questions arise from these results. Initially, one might wonder whether the use of learning is necessary for such a basic task. Indeed, a simple hand-coded attacker such as:

IF (state == 1) then execute **PassBall()**
ELSE execute **ShootToGoal()**

would perform even better than the Q-learning attacker since it would be 100% successful. Although the hand-coded attacker would perform well in such a simple task,

its performance degrades rapidly as the complexity of the problem increases. For example in the full 11 versus 11 robotic football challenge there are 23 mobile objects (22 agents and the ball) placed on a 105 by 68 grid. The (x,y) co-ordinates of the objects used alone provide with:

$$^{105\times68}C_{23} = {}^{7140}C_{23} = \frac{7140!}{23!\times(7140-23)!} = \frac{7140!}{23!\times7117!}$$

possible configurations (or states) to arrange all agents and the ball on the field. If the agents' velocities together with their body and neck facing directions are taken into account, the state space becomes even larger (ignoring the fact that all (x,y) co-ordinates are reported to an accuracy of 1 decimal place). Adding the real-time requirements of the agent's decision making process makes use of hand-coded action selection methods infeasible. In fact, the next chapter demonstrates how a learned policy outperforms a hand-coded one even on a much simpler (than the 11 versus 11) task.

On the other hand, given such a huge state space, it is reasonable to question the speed of convergence of Q-learning given its back-propagating nature. Since learning propagates backwards from the goal state, it would take an agent numerous iterations to learn a task in which the initial states and the goal states were far apart. However, by introducing intermediate *local goals* through *distribution of responsibilities*, each agent will only need to achieve a local goal that is dependent on its role. As mentioned earlier, a goalkeeper need not worry about how to score a goal, a task numerous stages away from its normal responsibilities. Instead, the goalkeeper's local goal should be to make a successful pass to a nearby defender. By linking these intermediate local goals a form of co-operation *emerges* in which each individual team member carries out a specific task towards the overall global goal. Although in RoboCup the ultimate goal is the score

against the opposition, this is not, and should not be treated as the aim of every individual agent. An example of this can be seen in figure 5.5.

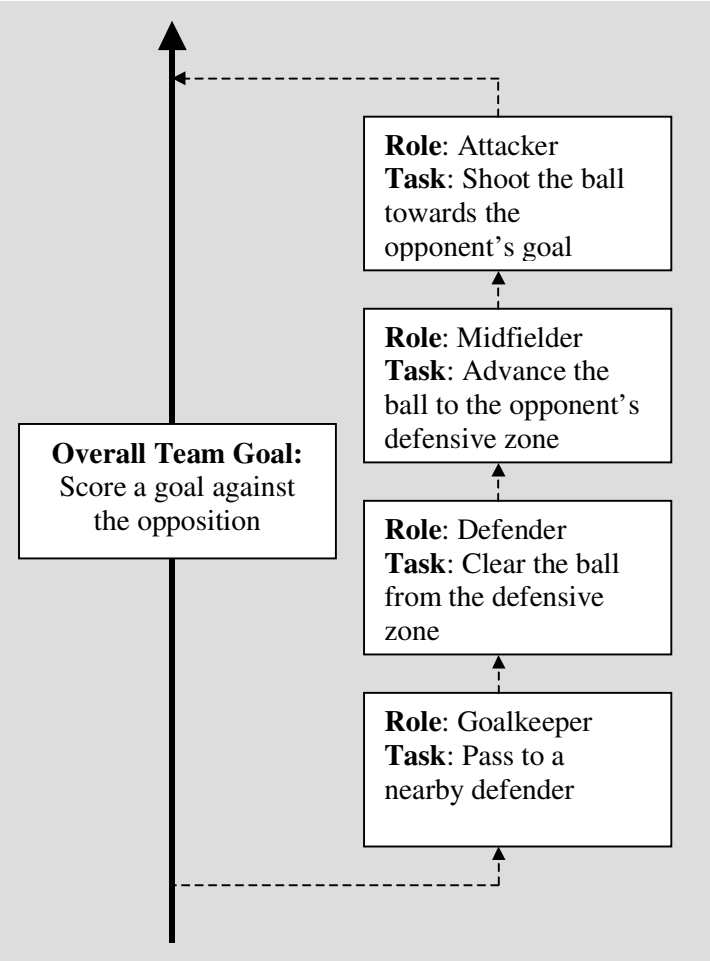


Figure 5.5: Achieving a global goal by distribution of responsibilities

The aim of this chapter was to show that *learning to act* using RL methods is possible in the RoboCup and similar domains. This was an important first step since the environment does not provide a reinforcement signal and the agent has to use *observational self-reinforcement* based on noisy sensations. In addition, this chapter has demonstrated how *co-operative* behaviour can *emerge* using *learning* even when the agents have no knowledge of the concept of co-operation. The basic task of 2v1 is one of the simplest co-operative tasks and was a necessary first step towards this investigation.

The scalability of this idea is tested with a more complex challenge (involving 5 agents) in the following chapter.

5.8 Related Work

This chapter demonstrated how RL methods (in particular Q-learning) could be used as a decision-making mechanism in the RoboCup and similar domains. In addition, the simple 2v1 challenge has demonstrated the ability of an agent to exhibit co-operative behaviour via its decision-making module, even when it has no knowledge of the concept of co-operation. This was the second and final step towards the development of KaBaGe-RL, which is presented in the next chapter.

Since RL in general is a large research topic, this section mainly focuses on the use of RL in multi-agent domains. Section 5.3 is based on work presented in [Mitchell, 1997] [Kaelbling et al., 1996] [Sutton and Barto, 1998] who also outline some of the advantages of RL methods (section 5.2). The 2v1 challenge described in section 5.5 was originally introduced by [Noda et al., 1996] who experimented with back-propagating neural network learning, followed by C4.5 decision tree learning [Noda et al., 1998]. Useful material on RL can be found in [Bertsekas and Tsitsiklis, 1996] and [Sutton and Barto, 1998]. Equally useful are [Sutton, 1992] and [Kaelbling, 1996], two special issues of the Machine Learning journal that focus on RL. Popular surveys on RL methods include those of [Barto, 1995] [Kaelbling et al., 1996] and [Keerthi and Ravindran, 1997]. The remainder of this section describes relevant RL applications, mainly in a multi-agent context.

[Littman, 1994] described a Q-learning like algorithm for finding optimal policies and demonstrated its application to a simple 2-player game in which the optimal policy was probabilistic. More specifically, the minimax-Q learning algorithm was introduced and tested using a simple 2-player zero-sum Markov game modelled after the game of soccer. Minimax-Q is an algorithm almost identical to Q-learning but with the *max* operator in the update step replaced by a *minimax* operator that can be evaluated by solving a linear problem. Note that the use of linear programming in the innermost loop of a learning algorithm is somewhat problematic since the computational complexity of each step is large and normally many steps will be needed before the system reaches convergence. The author argues that in games such as checkers, tic-tac-toe, backgammon, and go this kind of implementation is feasible. However, for more complex domains like RoboCup, such computational complexity makes minimax-Q learning infeasible (especially when one considers the real-time requirements of the decision-making process).

[Hu and Wellman, 1998] extended on Littman's work on zero-sum games to general-sum stochastic games. They proposed a multi-agent Q-learning method and proved that it converges to a Nash equilibrium under specific conditions. The proposed algorithm is useful for finding the optimal strategy when there is a unique Nash equilibrium in the game. However, when there are multiple equilibria, the algorithm must be combined with other learning techniques in order to find optimal strategies. The proposed algorithm does not require agents to know other agents' payoff function. However, it is required that agents can observe other agents' immediate payoffs and actions taken previously. Therefore the proposed algorithm is unsuitable for the problems addressed in this thesis.

[Claus and Boutilier, 1998] investigated the use of Q-learning for co-operative MAS. In doing so, they introduce two classes of learning agents, namely, independent learners and joint action learners. Independent learners apply Q-learning in the classic (single-

agent) sense ignoring the existence of teammates. On the other hand, joint action learners learn the value of their own actions in conjunction with those of other agents via integration of RL with equilibrium (or co-ordination) learning methods [Fudenberg and Levine, 1993] [Kalai and Lehrer, 1993]. In addition, [Claus and Boutilier, 1998] examined the influence of partial observability on joint action learners, and how game structure and exploration strategies influence the dynamics of the learning process and the convergence to equilibrium. Experimental results show that independent learners co-ordinate quite quickly. Note that the notion of co-operation between independent learners is directly related to the notion of emergent co-operation presented in this thesis.

[Tan, 1993] investigated the trade-offs between a group of independent RL agents and a group of co-operative RL agents (assuming both groups have the same number of agents). [Tan, 1993] considered 3 possible ways of co-operation between RL agents. First, agents communicate instantaneous information such as sensations, actions, or rewards. Second, agents communicate sequences of (sense, act, reward) triples. Third, agents can communicate learned decision policies. The basic principle is that if co-operation is carried out in an intelligent manner, each agent can benefit from other agents' instantaneous information, episodic experience, and learned knowledge. Three case studies are used in order to evaluate the performance of independent versus co-operative RL agents. All tasks involve hunter agents seeking to capture randomly moving prey agents in a 10 by 10 grid world. Although results are encouraging, this approach relies on heavy communication between team members. Implementing such an approach within RoboCup and similar domains is not possible, as there are very limited communication opportunities. In addition, this thesis contributes a non-communicating MAS where co-operation emerges through intelligent action selection rather than complex communication protocols.

[Sandholm and Crites, 1996] presented an empirical study of RL in the iterated prisoner's dilemma (IPD). They investigated the ability of a variety of Q-learning agents to play the IPD game against an unknown opponent. In some cases the opponent is another Q-learning agent whereas in others the opponent follows a fixed tit-for-tat strategy. Experimental results show the Q-learners to be consistently better. However, when competing against another learner, the non-stationarity of the environment has to be taken into account. The learners varied along three dimensions, namely, the length of history they received as context, the type of memory they employed (lookup tables based on restricted history windows or recurrent neural networks), and the exploration schedule they followed. Further experimentation revealed that agents with longer history windows, lookup table memories, and longer exploration schedules fared best in the IPD games.

[Crites and Barto, 1996] described the application of RL to the challenging real-world problem of elevator dispatching. Their implementation considers a team of RL agents, each of which is responsible for controlling one elevator car. The team receives a global reinforcement signal that appears noisy to each agent due to the effects of the actions of the other agents, the random nature of the arrivals and the incomplete observation of the state. To evaluate the performance of the RL agents, [Crites and Barto, 1996] considered two different architectures, a parallel and a fully decentralised. In the parallel architecture, the agents share a single network, allowing them to learn from each other's experiences and forcing them to learn identical policies. In the fully decentralised architecture, the agents have their own networks, allowing them to specialise their control policies. In either case, agents do not have explicit access to the actions of the other agents. Co-operation has to be learned indirectly via the global reinforcement signal. Each agent faces added stochasticity and nonstationarity because its environment contains other learning agents. Both architectures were compared to a number of heuristic

algorithms including dynamic load balancing, longest queue first, and finite inter-visit minimisation. Both RL systems achieved very good performance. In addition, the decentralised RL system was able to achieve as good a level of performance as the parallel RL system. This is a significant observation as it shows that decentralised agents can learn to co-operate via their decision-making mechanism. This approach comes very close to the idea of *learning to co-operate* presented in this thesis. However, good performance of such a learning scheme to more complex domains like RoboCup is not guaranteed. Note that the elevator-dispatching scenario does not consider adversaries and agents' perceptions and actions are not noisy.

[Lauer and Riedmiller, 2000] considered distributed RL in co-operative multi-agent-decision-processes, where a group of simultaneously and independently acting agents tries to maximise the discounted sum of rewards. Each agent has no information about its teammates' behaviours. Therefore, each agent has to consider its teammates' actions in order to find an efficient co-operative policy. To that extend, [Lauer and Riedmiller, 2000] proposed a model-free distributed Q-learning algorithm for co-operative multi-agent-decision-processes. This algorithm allows independent learners to derive an optimal policy in a co-operative multi-agent Markov decision process (MAMDP) without additional information or communication between the agents. Although they provide encouraging results for deterministic environments, they conclude that the possibility of using the proposed distributed RL in stochastic environments remains an open question.

Within the RoboCup domain a number of teams have investigated the use of RL methods. Initially, [Andou, 1998] used RL methods for positioning of the agents on the field. The proposed method of *observational RL* allowed agents to notice where the ball has travelled most often in the past and to adjust their positions so that they can be closer to the ball's path in the future.

Also, [Riedmiller, et al., 2001] described how RoboCup can be modelled as a MAMDP and also outline how optimal policies can be defined in such a domain. Their implementation reduces the action space by combining primitive actions and forming complex behaviours, which they call *macro actions* or *moves*. These *moves* are learned using RL with neural network based function approximators [Riedmiller, 2000]. Initial work involves experimentation with attacking behaviours on a 3 versus 4 scenario. Results indicate an improvement in performance by a factor of 3.

Chapter 6

Learning to Co-operate Using KaBaGe-RL

6.1 Overview

An equally important challenge to that of *learning-to-act* in multi-agent systems is the one of *co-operation*, especially in adversarial domains like RoboCup. It has been shown that groups of agents working together can derive more efficient solutions in terms of energy, costs, time and quality [Kube and Zhang, 1992] [Noreils, 1993] [Altenburg, 1994] [Mataric, 1995]. The distribution of responsibilities and multiple roles in co-operative frameworks is a common feature. Each agent in a team has an individual role and therefore a set of responsibilities [Ch'ng and Padgham, 1997a] [Ch'ng and Padgham, 1997b]. The aim of this chapter is not to present a new co-operative framework. Instead, the purpose is to show that an intelligent agent working in a team should be able to *learn* that *co-operation* is beneficial. This chapter demonstrates how co-operation *emerges* through an agent's decision making process, even when the agent has no knowledge of the concept of co-operation.

Chapter 4 demonstrated how Kanerva Based Generalisation (KaBaGe) could be used as a function approximator in problems with large state spaces. Chapter 5 used the simple 2v1 co-operative task to show how tabular reinforcement learning (RL) based action selection methods can learn optimal policies, which involve co-operative behaviour, in an

environment that does not provide the agents with a reinforcement signal. The 2v1 challenge is one of the simplest co-operative tasks and was a necessary first step towards the development of KaBaGe-RL. To that extend, this chapter combines KaBaGe and RL in order to construct a decision-making module that is suitable for more complex tasks involving much larger state spaces. It will be shown that KaBaGe-RL learns to outperform a number of benchmark policies, including a hand-coded one. In addition, the results presented in this chapter demonstrate how an agent *learns to co-operate* in order to achieve a global goal even when it has not been explicitly programmed to do so. Parts of this chapter have originally appeared in [Kostiadis and Hu, 2001].

6.2 Why KaBaGe-RL

As explained in earlier chapters, on large problems, RL systems cannot rely on the use of lookup tables in order to maintain an accurate representation of the environment. In the past, a number of systems have combined RL based decision making modules with function approximators in order to generalise between similar situations and actions. However, [Boyan and Moore, 1995] reported a series of negative results in attempting to do so even on simple control problems with continuous state spaces. More specifically, they considered extending discrete value iteration to the continuous case by replacing the lookup table over all states with a function approximator trained over a sample of states. The hope is that these intermediate value functions will be smooth and will achieve convergence. Unfortunately, experimental results revealed that divergence occurs frequently even for quite simple problems.

On the other hand [Sutton, 1996] presented positive results for all the control tasks attempted by [Boyan and Moore, 1995], and for one that is significantly larger. The most

important differences are that [Sutton, 1996] used sparse-coarse-coded function approximators (rather than global ones) and also learned online (rather than offline). In addition, [Tesauro, 1992] [Crites and Barto, 1996] presented very effective practical applications of RL methods combined with function approximators.

The key requirements for a given method to obtain good performance are not yet fully known. However, this chapter presents empirical evidence that demonstrates how Kanerva-coding based function approximation together with residual Q-learning can be very effective even on complex domains such as RoboCup. To the best of the author's knowledge, there have been no attempts to combine Kanerva's Sparse Distributed Memory (SDM) model together with RL methods. This, together with the results presented in previous chapters regarding the computational efficiency of KaBaGe, has been the main reason for the proposed KaBaGe-RL approach.

6.3 The 3 Attackers versus 2 Defenders Possession Challenge

The performance of KaBaGe-RL is tested using the *3 Attackers versus 2 Defenders* challenge (or 3v2 for short), a subtask of the RoboCup domain. The 3v2 challenge was originally introduced by [Stone et al., 2001] in what they termed the *keepaway* problem. The description that follows, is based on their original description presented in [Stone et al., 2001].

In the 3v2 possession challenge there is one team of *Attackers*, trying to maintain possession of the ball within a limited region, while another team, the *Defenders,* is trying to gain possession of the ball. Each episode starts with the *Attackers* having possession. As soon as the ball goes outside the region of play or the *Defenders* take possession of the

ball, the episode ends and the agents are reset for another episode (having possession of the ball means that the ball is within kicking distance from the agent). The experiments presented in this chapter use a region of 20m x 20m (figure 6.1).

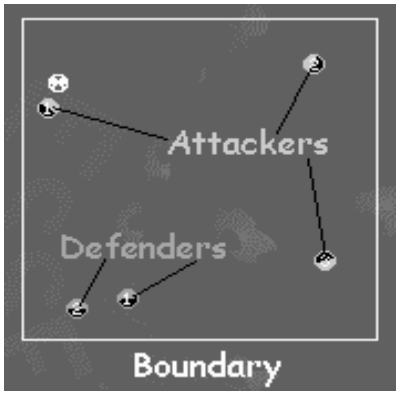


Figure 6.1: Screen shot from a 3v2 possession episode
in a 20m x 20m region.

An omniscient coach agent manages the play, ending each episode when the ball goes outside the region of play or when the defenders gain possession. The coach then resets the position of the ball and agents and starts the next episode. At the beginning of each episode both defenders are placed semi-randomly within a certain distance from the bottom-left corner of the play region. The attackers are then placed in the same manner, one in each of the remaining 3 corners of the play region. The ball is placed close to the attacker of the top-left corner. Figure 6.1 shows a screen shot of an initial configuration for an episode.

All the experiments were done using version 6.07 of the standard RoboCup simulator [Noda et al., 1998]. To concentrate on the efficiency of the decision-making process of each agent, a simplification was made to the original version of the simulator. In the original version, each agent has a limited field of view (by default each agent can only see objects within a 90-degree view cone), and the precision of an object's sensed location

degrades with distance. For these experiments the agents are equipped with a full 360-degree noise free vision (actions retain their original noisy characteristics). It is not yet clear whether this simplification is necessary or helpful.

Recall that the RoboCup simulator operates in discrete time steps, t = 1, 2, …, n each representing 100ms of simulated time. As soon as one episode ends (either when the ball goes outside the region of play or when the defenders gain possession), another begins. Each agent is presented with a sequence of these episodes. This can be seen as an instance of the discrete-time, episodic RL framework. As stated earlier, the defenders' task is to gain possession of the ball, while the attackers try to maintain possession for as long as possible. For this task, each agent is equipped with the following high-level behaviours:

- **HoldBall()** – Remain stationary while keeping the ball in a position that is as far away from the opponents as possible.
- **PassBall(t)** – Kick the ball directly towards teammate t.
- **InterceptBall()** – Intercept a moving ball or move towards a stationary ball as fast as possible.
- **GetOpenToReceive()** – Move to the least congested position within the region of play in order to receive a pass.

The defenders' policy space is quite simple. Each defender always executes **InterceptBall()**. This is necessary since two defenders are required in order to steal the ball from an attacker. If only one defender attempts to steal the ball, the attacker can execute the **HoldBall()** behaviour, and maintain possession for a long period of time.

Once the ball gets within a defender's kicking range the episode ends. The attackers' policy space is more complex and can be seen in figure 6.2.

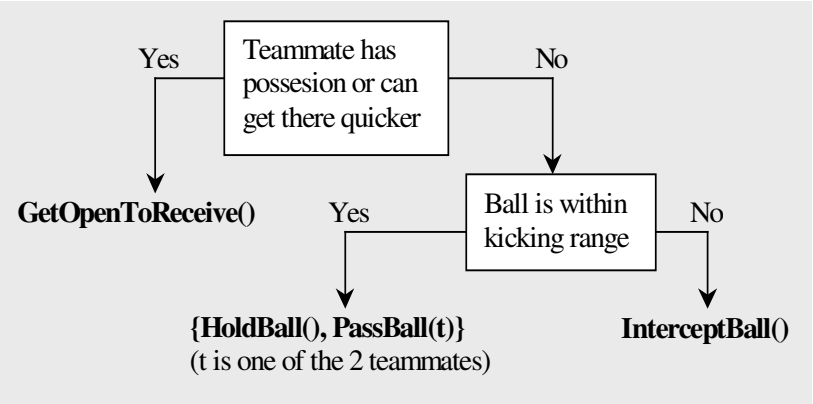


Figure 6.2: The attacker's policy space.

The most important decision point for an attacker occurs when the ball is within its kicking range. This is the moment when the agent has to decide whether to hold the ball, pass to teammate 1 (nearest teammate), or pass to teammate 2 (furthest teammate). For this subset of the attackers' policy space (i.e. {**HoldBall()**, **PassBall(Teammate 1)**, **PassBall(Teammate 2)**}), residual Q-learning has been applied in order to optimise the decision making of the agent and maximise the possession time.

Possession football is a subtask of robotic football. Since there are fewer agents involved and the game takes place in a smaller region, each agent can concentrate on the same high-level goal without the need to balance offensive and defensive considerations. Nevertheless, learning to maintain possession of the ball in a smaller region is of great importance in full-scale 11 vs. 11 games, and is one of the first things human players learn to do.

6.4 Implementing KaBaGe-RL

The 2v1 challenge used in the previous chapter could be fully represented using 4 states and 2 actions. For such a simple task there is no need for generalisation as the whole environment can be represented using a table with each table entry corresponding to a unique state-action pair. However for the 3v2 possession challenge a tabular representation is not practical. In a region of 20m × 20m, the (x,y) co-ordinates of the agents and the ball used alone provide with:

$$^{20\times 20}C_6 = {}^{400}C_6 = \frac{400!}{6!\times(400-6)!} = \frac{400!}{6!\times 394!}$$

possible configurations (or states) to arrange all agents and the ball within the play region. If the agents' velocities together with their body and neck facing directions are taken into account, the state space becomes even larger (ignoring the fact that all (x,y) co-ordinates are reported to an accuracy of 1 decimal place). Even if the memory requirements could be met to construct such a table, the data needed to fill it accurately would take a long time to obtain. As a result, reaching convergence to an optimal policy would also take a long time. In addition, such an implementation would make inefficient use of experience. One would expect similar states to have similar optimal actions.

To apply RL methods in such a problem, the use of a function approximator is necessary. A number of systems have combined function approximation and RL in the past [Zhang and Dietterich, 1995] [Shewchuk and Dean, 1990] [Miller et al., 1994] [Lin and Kim, 1991] [Watkins, 1989] [Rummery and Niranjan, 1994]. To the best of the author's knowledge, there have been no attempts to combine Kanerva's SDM together with RL.

6.4.1 KaBaGe and Feature Representation

Recall that the efficiency of KaBaGe is directly related to the feature representation scheme. A good candidate would be a scheme that encapsulates all the useful information needed by an agent in order to make a decision. In addition, for the 3v2 challenge, the chosen scheme must be able to recognise symmetric configurations of the agents and the ball. For example, an agent that successfully learned to choose optimal actions when positioned near the top-left corner should be able to use this experience when positioned near any of the remaining three corners. Therefore, the chosen scheme should not be based on the positions of the agents and the ball but rather on the distances and angles between them. Such a scheme already exists and was originally used by [Stone and Sutton, 2001] in a CMAC implementation. According to [Stone and Sutton, 2001] the environment is encoded using the following 13 state variables:

1. Distance(A1, C)
2. Distance(A1, A2)
3. Distance(A1, A3)
4. Distance(A1, D1)
5. Distance(A1, D2)
6. Distance(A2, C)
7. Distance(A3, C)
8. Distance(D1, C)
9. Distance(D2, C)
10. Minimum(Distance(A2, D1), Distance(A2, D2))
11. Minimum(Distance(A3, D1), Distance(A3, D2))

12. Minimum(Angle(A2, A1, D1), Angle(A2, A1, D2))

13. Minimum(Angle(A3, A1, D1), Angle(A3, A1, D2))

where:

- **C** – is the centre of the region of play
- **A1** – is the attacker in possession of the ball (ball shown as a dark circle in figure 7.3)
- **A2** – is the closest teammate (attacker) to A1
- **A3** – is the second closest teammate (attacker) to A1
- **D1** – is the closest opponent (defender) to A1
- **D2** – is the second closest opponent (defender) to A1
- **Distance(A1, C)** – is the distance between attacker 1 and the centre of the play region
- **Angle(A2, A1, D1)** – is the angle shown in figure 7.3.

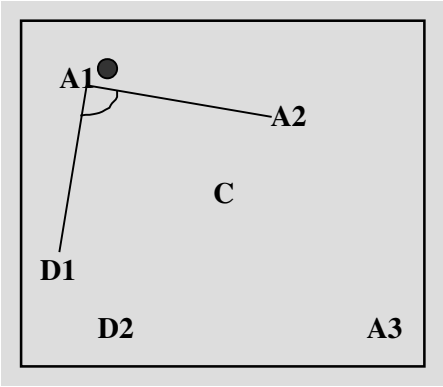


Figure 6.3: The state variables used by KaBaGe.

Recall that Kanerva-coding only deals with binary features. However, the distances and angles used are not in a binary format. As mentioned in chapter 4, the first step in the KaBaGe module is that of input conversion if the input is not in a binary format. The

method used converts all variables into 3-bit binary features. Out of these bits only one is set to 1 and the remaining two are set to 0. For distances those bits represent how close the two objects are. If the objects are very close the first bit is set and the feature appears as (1, 0, 0). Similarly if the two objects are too far apart the last bit is set and the feature appears as (0, 0, 1). If the two objects are neither too close nor too far, then the middle bit is set and the feature appears as (0, 1, 0). In the current implementation too close indicates a distance less than 3 meters away and too far indicates a distance more than 8 meters away. Similarly for the angles, too narrow indicates an angle less than 20 degrees and too wide indicates an angle more than 60 degrees. These values may not be optimal and were based on a simplified statistical analysis after gathering all possible values for each feature from 10000 trials. A frequency graph for each state variable was used to estimate the possible partitioning points.

The core of the KaBaGe module consists of 5000 prototypes, which have 39 bits each (since there are 13 features of 3 bits each). Each of the 5000 prototypes is connected to each of the possible actions as shown in figure 6.4 resulting to 15000 links.

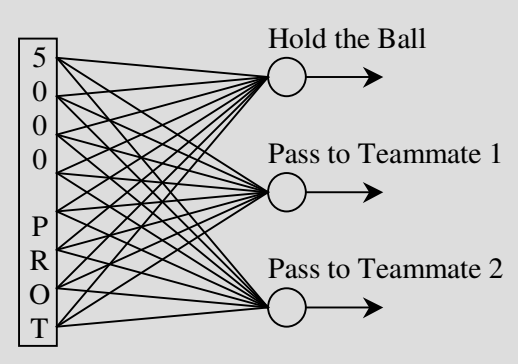


Figure 6.4: KaBaGe links to all possible actions

(5000 prototypes to each of the 3 actions).

This input conversion already provides some form of generalisation by partitioning continuous state variables. The possible number of states has been reduced to 3^{13}=1594323, which is far less than the value computed earlier in this section. However, out of 1594323 possible states, KaBaGe attempts to generalise by only using 5000 randomly constructed prototypes, i.e. 5000 randomly selected states.

6.4.2 Q-learning and Residual Algorithms

Q-learning is guaranteed to converge to the optimal solution when used with lookup tables. However, [Baird, 1995] showed that it can become unstable when implemented together with a function approximation system. Indeed, results of such implementations have been mixed. Successful applications include Samuels' checkers [Samuels, 1959], Tesauro's backgammon [Tesauro, 1990], and Lin's robot navigation [Lin, 1992]. However these algorithms are extremely unstable as demonstrated in [Gordon, 1995]. [Boyan and Moore, 1995] presented several simple situations where popular algorithms fail miserably. Some reasons for these failures were given in [Thrun and Schwartz, 1993] and [Sabes, 1993]. Baird presented a class of algorithms, *residual algorithms*, that guarantee convergence of Q-learning (and other RL algorithms) even when used together with a function approximator. A brief description of these algorithms is given in this section, which is based on [Baird, 1995].

Direct Algorithms

A more abstract version of the Q-learning update rule presented in chapter 6 can be expressed as follows:

$$V(s_t) \xleftarrow{a} R + \gamma V(s_{t+1}) \tag{6.1}$$

where $V(s_t)$ represents the value function for state s at time t and $\alpha \in (0,1)$ is the learning rate. Under certain assumptions, the update rule (6.1) is guaranteed to converge to the optimal value function as the learning rate decays to zero at an appropriate rate [Watkins and Dyan, 1992]. These assumptions include the task having a finite number of states, each $V(s)$ to be represented by a unique entry in a lookup table, and each possible transition to be experienced an infinite number of times. Even if $V(s)$ was represented by a function approximator other than a lookup table, update (6.1) could be implemented directly by combining it with the backpropagation algorithm [Rumelhart et al., 1986]. After an action was chosen for a given state, the reward from that action would be used to adjust all the weights through gradient descent and make the actual output shift closer to the desired output. For any particular weight in the function approximator, the weight change would be given by:

$$\Delta w_d = a(R + \gamma V(s_{t+1}) - V(s_t)) \frac{\partial V(s_t)}{\partial w_d} \tag{6.2}$$

Equation (6.2) is exactly the *TD(0)* algorithm [Sutton and Barto, 1998]. According to [Baird, 1995] it is called the *direct* implementation of incremental value iteration. The direct algorithm reduces to the original algorithm when used with a lookup table. Since the convergence is guaranteed for lookup table implementations, one might also expect it to converge for general function approximation implementations. However, [Baird, 1995] demonstrated this is not the case.

Residual Gradient Algorithms

It is unfortunate for a RL algorithm that is guaranteed to converge for lookup tables to be unstable when combined with a function approximator. [Baird, 1995] proposed a new class of algorithms, *residual gradient algorithms*, that are guaranteed to converge even when used with function approximators. For a given value function V, and a given state s, the *Bellman residual*, is defined as the difference between the two sides of the Bellman equation. The *mean squared Bellman residual* for a Markov Decision Process with n states is therefore defined to be:

$$E = \frac{1}{n}\sum_{s}[\langle R + \gamma V(s_{t+1})\rangle - V(s_t)]^2 \qquad (6.3)$$

If the Bellman residual is nonzero, the resulting policy will be sub-optimal. However, for a given level of Bellman residual, the degree to which a policy yields sub-optimal reinforcement can be bounded [Williams and Baird, 1993]. Therefore, it should be possible to change the weights of a function approximator by performing gradient descent on the mean squared Bellman residual, E. In other words, after a transition from state s_t to state s_{t+1}, with reinforcement R, a weight w would change as follows:

$$\Delta w_{rg} = -a[R + \gamma V(s_{t+1}) - V(s_t)][\frac{\partial}{\partial w_{rg}}\gamma V(s_{t+1}) - \frac{\partial}{\partial w_{rg}}V(s_t)] \qquad (6.4)$$

For a system with a finite number of states, E is zero if and only if the value function is optimal. Therefore, performing gradient descent on E guarantees that E will eventually converge to a local minimum.

Residual Algorithms

Although direct algorithms can learn very fast, they are unstable. On the other hand residual gradient algorithms are stable but can be quite slow. Direct algorithms attempt to make each state match its successors, but ignore the effects of generalisation during learning. Residual gradient algorithms take into account the effects of generalisation but attempt to make each state match both its successors and its predecessors. Combining both algorithms (equations 6.2 and 6.4) gives raise to a new class of algorithms, *residual algorithms*, which are guaranteed to converge for an appropriate choice of ϕ, and where the weight change is given by:

$$\Delta w = (1-\phi)\Delta w_d + \phi \Delta w_{rg} \tag{6.5}$$

where Δw_d and Δw_{rg} denote the weight changes for the direct and residual gradient algorithms respectively. Given the stability of residual algorithms and their ability to learn quickly (given an appropriate choice of ϕ), all the experiments presented in this chapter use *residual Q-learning* as an action selection method. Therefore, all the weights are updated using the following rule:

$$\begin{aligned}\Delta w_r = -\alpha(R + \gamma \max_{u_{t+1}} Q(s_{t+1}, u_{t+1}) - \\ Q(s_t, u_t))(\phi\gamma \frac{\partial}{\partial w} \max_{u_{t+1}} Q(s_{t+1}, u_{t+1}) - \frac{\partial}{\partial w} Q(s_t, u_t))\end{aligned} \tag{6.6}$$

where

$$Q(s_t, u_t) = R + \gamma \max_{u_{t+1}} Q(s_{t+1}, u_{t+1}) \tag{6.7}$$

The weights of all the prototypes that are active for a particular state are added in order to form the Q value for each action. The action with the highest sum (i.e. the highest Q value) is the one that is executed. Ties are broken according to an arbitrary ordering of the 3 possible actions. Optimistic action initialisation (as explained in the previous chapter) ensures that all possible actions will be tried at least once.

6.5 Experimental Results

The performance of KaBaGe-RL was evaluated by comparing it against the following five benchmark policies:

- **AlwaysHold()** – Always hold the ball, never passes.
- **AlwaysPassNear()** – Immediately passes to the nearest teammate.
- **AlwaysPassFar()** – Immediately passes to the furthest teammate.
- **Random()** – Randomly execute one of the 3 possible actions.
- **HandCoded()** – Defined as follows:

IF (no defenders within 10 meters) THEN **HoldBall()**
ELSE IF (teammate t in better position AND pass is likely to succeed)
THEN **PassBall(t)**
ELSE **HoldBall()**

Notice that the first three benchmark policies correspond to the three actions in the attackers' policy space. Each policy was used for 19000 trials (or episodes). The results (i.e. the possession times) were then averaged in bins of 1000 episodes and can be seen in

figure 6.5. Recall that a cycle is 100msec, so for example 100 cycles correspond to 10 seconds. KaBaGe-RL clearly outperforms all other policies even after the first 1000 trials. Also note that KaBaGe-RL reached its near optimal performance of 98 cycles (9.8 seconds) after only 4000 trials.

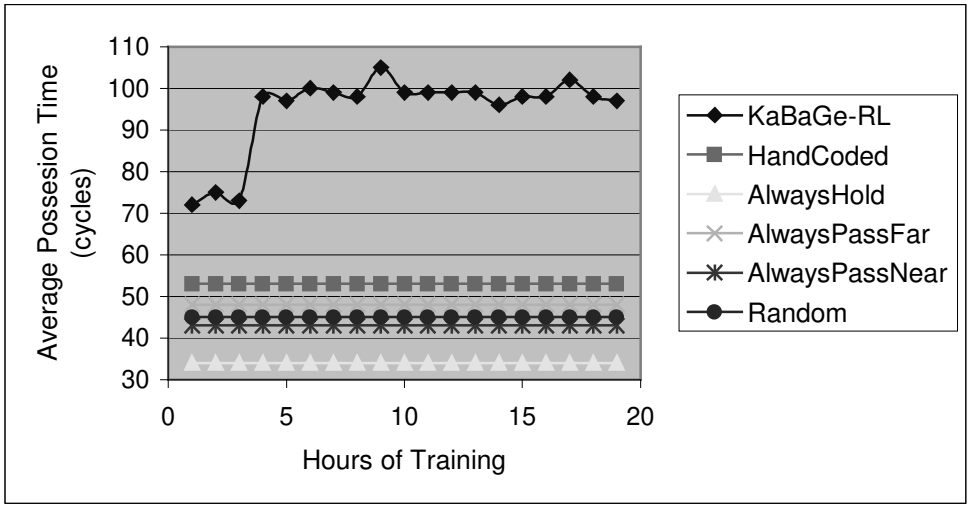


Figure 6.5: Possession time versus hours of training

(bins of 1000 episodes).

During these experiments, KaBaGe-RL was configured with γ=0.99 and ϕ=0.4. Another important parameter is the threshold of the KaBaGe module. Recall that the environment was encoded using 13 state variables. Also recall that a prototype gets activated if the number of features that are present in the input signal exceeds a certain threshold. In this case the threshold was set to a value that required at least 7 features (i.e. slightly more than half) to be present in order for a prototype to be activated. Tests requiring either 6 or 8 features to be present performed significantly worse. This was due to a large number of prototypes being always active (when 6 features were used) or being always inactive (when 8 features were used).

One important question is how to chose suitable values for the variables γ and ϕ. Clearly there are a number of different combinations one can try since both γ and ϕ are real numbers between 0 and 1. Fortunately [Baird, 1995] presented a way of calculating ϕ so that learning can be as fast as possible without causing the weights to blow up. Recall that ϕ affects mostly the speed of learning and not its quality (assuming it is within a legal range). On the other hand γ directly affects the quality of learning since it dictates the importance of future rewards. However, it was immediately clear that when γ was less than 0.93 (with ϕ fixed to 0.4) the performance of the system degraded rapidly. This is presumably because the benefits of passing were too far in the future, causing agents to adopt a more selfish behaviour (i.e. hold the ball) rather than a co-operative one (i.e. pass the ball). It is worth mentioning that [Stone et al., 2001] reported similar results for a Q-learning implementation that used a CMAC based function approximator.

6.6 Related Work

This chapter demonstrated how KaBaGe and RL (in particular residual Q-learning) could be combined in order to produce a decision-making module for noisy, real-time, collaborative and adversarial domains that involve large state spaces. The *3v2 Possession* challenge demonstrated the ability of an agent to exhibit co-operative behaviour via its decision-making module even when it has no knowledge of the concept of co-operation. Experimental results showed that KaBaGe-RL performs significantly better than the benchmark policies used. The *3v2 Possession* challenge described in section 6.3 was originally introduced by [Stone et al., 2001]. The feature representation scheme used in section 6.4.1 is due to [Stone and Sutton, 2001]. Finally the description of direct, residual gradient and residual algorithms presented in section 6.4.2 are based on [Baird, 1995],

who also outlined the residual Q-learning equations (6.6) and (6.7) that were used in the experiments presented in this chapter.

A number of systems have combined function approximation and RL in the past [Zhang and Dietterich, 1995] [Shewchuk and Dean, 1990] [Miller et al., 1994] [Lin and Kim, 1991] [Watkins, 1989] [Rummery and Niranjan, 1994]. For example, [Uther and Veloso, 1997] introduced an extension to prioritised sweeping, namely, *fitted prioritised sweeping*, that allows generalisation of learnt knowledge over neighbouring states in the domain. [Uther and Veloso, 1997] followed the results by [Boyan and Moore, 1996] which illustrated that generalising function approximators are not problematic if the approximation is not iterated, i.e. if an approximation is not used to update itself. Fitted prioritised sweeping uses the standard prioritised sweeping as a base but performs the generalisation afterwards. More specifically, at each time-step during the game, the agent updates a standard prioritised sweeping Q-function over a standard discretisation of the state space. However, in order to choose an action the agent reads the Q-values from a generalisation function approximator that was learnt during the previous game. At the end of each game, the Q-values that were generated by the prioritised sweeping algorithm for all visited state-action pairs is used to update the generalising function approximator. Experimental results show that fitted prioritised sweeping performs significantly better that non-generalising algorithms and is also guaranteed to converge.

Also, [Dean et al., 1992] investigated the use of CMAC and RL methods for planning and control. In doing so, they presented two different CMAC methods, namely, *vanilla* CMAC and *multi-resolution* CMAC. Vanilla CMAC uses multiple partitions of the input space that are offset from one another to form overlapping regions. Each of the partitions is mapped onto the set of storage elements with a different hashing function. For a given input, the output produced by vanilla CMAC is the average of contents of the storage

elements determined by hashing the set of regions containing the input. Multi-resolution CMAC (originally presented in [Moody, 1989]) uses a successive refinement strategy to achieve a useful trade-off between the speed and the accuracy of learning. To evaluate the performance of vanilla CMAC [Dean et al., 1992] used the *inverted pendulum balancing* learning task described in [Barto et al., 1983]. For these experiments the system only received negative reinforcement when the pendulum attained an angle of ±12°. Even so, the system learnt to balance the pendulum after only 38 failed attempts.

One of the main results of this chapter is that intelligent agents *learn to co-operate* even when they have no knowledge of the concept of co-operation. In the recent years, co-operation between autonomous intelligent agents has received a great deal of attention. [Doran et al., 1997] presented a useful study of the concept of co-operation in multi-agent systems (MAS). They addressed a number of general questions such as what co-operation is and what are its limits but also discuss specific issues including whether co-operation should be motivated a priori or whether it can emerge through complex social relationships. Answers to these questions varied from author to author. Franklin classified a MAS where each agent pursues its own agenda as *independent* [Franklin and Graesser, 1997]. A MAS is *discrete* if it is independent and the agendas of the individual agents bear no relation to one another. Discrete systems involve no co-operation. However, agents can co-operate with no intention of doing so like in the example of the puck gathering robots [Beckers et al., 1994]. Puck gathering is an *emergent* behaviour of the system since the agents appear to be working together from the viewpoint of an external observer, but from the agent's viewpoint they are not. They are simply carrying out their own individual behaviour. Doran offered a more tightly defined viewpoint on co-operation. He saw co-operation more as a property of the actions of the agents involved. Therefore, for a MAS like the one presented in this thesis where agents share a

common global goal, co-operation occurs when the agent's actions tend to achieve that goal. Note that this definition focuses on the actions of the agents and their goals, irrespective of how they arise. In other words, like Franklin's definition of *emergent co-operation*, Doran's work does not require agents to *intend* to co-operate for co-operative problem solving to occur. This view of *emergent co-operation* is the one adopted throughout this thesis and one of the main questions addressed was whether agents can *learn to co-operate* under this definition of co-operative problem solving.

[Bottazzi et al., 1999] investigated the collective outcomes of different learning procedures by heterogeneous agents who play *minority games*, a highly stylised metaphor of speculative market interactions. In the original game, introduced by [Challet and Zhang, 1997] [Challet and Zhang, 1998], a population of agents must each simultaneously and independently choose between two sides. The side chosen by the minority of the agents is the winning one, and agents who chose it are rewarded one point each. Agents of the losing side win nothing. Each agent is initially equipped with a fixed number of strategies, which are updated throughout the game according to a deterministic algorithm. Although this game is very simple, it does capture some basic features of speculation in financial markets whereby agents try to infer some structure in the history of collective interaction and through that *beat the market* i.e. beat the majority view. [Bottazzi et al., 1999] assumed a team of agents that can only communicate with each other through the process of buying and selling. They investigated the conditions under which repeated interaction amongst agents causes some form of aggregate self-organisation to spontaneously emerge.

[Sen et al., 1994] investigated the use of machine learning techniques as a means to provide effective co-ordination. Their work is directly related to the one presented in this thesis, as their framework does not allow for communication between the agents, and co-

operation emerges via intelligent decision making. In fact, in their experiments with the block-pushing task, the agents involved have no knowledge about each other. More specifically, [Sen et al., 1994] applied Q-learning on multiple agents that learn co-operative strategies without having to rely on shared information. Also, the agents involved are unaware of the capabilities of other agents. Previous attempts in using learning for co-operation between multiple agents rely on unrestricted or explicit information sharing [Sian, 1991] [Tan, 1993] [Weib, 1993]. In fact, agents independently optimise their own rewards, and global co-ordination between multiple agents emerges without explicit or implicit information sharing. Experimental results demonstrated the effects of varying learning rates and varying agent capabilities to the performance of the system. Results indicate that agents can learn co-operative behaviours without even being aware of each other. Although conceptually the approach adopted in this thesis is similar to [Sen et al., 1994], there are a number of differences both in the implementation and in the domain characteristics. For example, the block-pushing does not require a function approximator, there are no real-time constraints, and there are no adversaries.

[Abul et al., 2000] presented two multi-agent co-ordination mechanisms based on RL where agents do not require an explicit communication mechanism in order to learn co-operative behaviour. The first mechanism, *perceptual co-ordination mechanism* (PCM), assumes that teammates are included in state descriptions and co-ordination information is learned from state transitions. In other words, agents treat their teammates as part of the environment with their behaviours embedded in state transitions. Note that this method is similar to single-agent learning because agents are unaware of teammates in their cognition. The second mechanism, *observing co-ordination mechanism* (OCM) is similar to PCM with the additional property that rewards of nearby agents are observed from the environment. The agent's own reward together with these observed rewards are

then used to construct an optimal policy. Note that OCM tends to increase region-wide joint rewards. The idea behind OCM is that the sum of rewards of agents in a region forms their joint actions' utility. Therefore, agents will learn to co-operate if every agent in the region tries to increase this sum. To evaluate the performance of these mechanisms [Abul et al., 2000] experimented with the *adversarial food-collecting world*. Results showed that both the PCM and OCM methods performed well.

Within the RoboCup domain there has been some attempts to implement co-operation using pre-arranged fixed plans. For example [Pfeifer and Wainer, 2000] investigated the use of pre-set multi-agent plans as a tool to design co-operative behaviour between groups of agents. They claimed that in an environment like RoboCup some situations are recurrent and therefore a domain expert can exploit such similarities. In other words, when similar situations occur, the pre-set plan can be executed by a group of agents possibly yielding the projected goal. There are a number of advantages in such an approach. For example, the computational resources needed to achieve a similar complexity of behaviour as the pre-set plan could not be available due to the real-time constraints. In addition, a generic strategy would hardly analyse the situation as deeply as the domain expert that projected the plan would. However, competition experience has shown that such pre-set co-ordination is very fragile and is more likely to fail than succeed. Such pre-planned, domain specific co-ordination normally fails to provide the flexibility that is required in complex domains. Common reasons for failure include inconsistent world state information between teammates, adaptation from opponents and inability to meet timing requirements.

Similar is the work of [Stone and Veloso, 1999a] who argued that most proposed teamwork structures (e.g. joint intentions [Cohen et al., 1998], shared plans [Grosz, 1996]) rely on heavy communication and negotiation between the agents in a MAS and

are therefore not suitable for dynamic, real-time domains. Their work investigates periodic team synchronisation domains, where agents in a team alternate between periods of limited and unlimited communication. For example, in the RoboCup domain, teams can plan strategies before the game, at halftime, or at other breakpoints. However, during the course of the game communication is limited. During the periodic synchronisation opportunities, certain multi-agent plans are formed (which [Stone and Veloso, 1999a] term *locker-room agreements*) that are remembered identically by all agents in the team. This common knowledge is then used in order to co-operate efficiently during the periods of limited communication. Although this approach has demonstrated good performance during competitions, it still relies on communication during the game. Also, due to the large amount of hidden state and the inaccurate visual information, agents in the team cannot always determine their roles in the pre-agreed plan. Nevertheless, [Stone and Veloso, 1999a] offered a flexible teamwork framework for domains with limited communication opportunities.

On the other hand, [Stolzenburg et al., 2000] combined the advantages of procedural and logic programming in order to create a hybrid layered architecture for the RoboCup domain. Their implementation consists of 4 layers, ranging from low-level basic skills to multi-agent co-operative behaviour, and is done using a hybrid of C++ and an enhanced version of ECLiPSe-Prolog. Part of their investigation questions whether co-operation should be invoked explicitly by communication or whether it is sufficient and more robust to have emergent co-operation. Initial attempts exploit the common knowledge between the agents (since all teammates are essentially instances of the same program). A co-operative plan becomes an acyclic graph of desires with a designated start node. Each agent in the plan has a specific role and a specific task within the global plan. The pre-programmed common knowledge is used to determine each agent's role. However, since

such a framework relies on local observations, it is quite difficult for agents to simultaneously identify their roles within a plan. Also, due to the large amount of hidden state in the RoboCup domain, agents may not always be aware of nearby teammates. To tackle this problem [Stolzenburg et al., 2000] recommend the use of communication between teammates. This way, co-operative partners could broadcast their co-ordinates or even their whole belief states.

Similar to KaBaGe-RL is [Stone and Veloso, 1999b] who presented a multi-agent learning paradigm, namely, team-partitioned opaque-transition reinforcement learning (TPOT-RL). Like KaBaGe-RL, TPOT-RL enables teams of agents to learn effective policies with very few training examples even in problems that have large state spaces with large amounts of hidden state. TPOT-RL maps states to actions using a 3-step process. Firstly it generalises the input using action dependent features. Secondly, the feature vector produced by the generalisation is used to estimate the expected reward of taking each possible action using a learned value function. Finally, an action is chosen for execution. One of the main differences between KaBaGe-RL and TPOT-RL is in the generalisation process. TPOT-RL does not use a function approximator but instead evaluates each possible action locally based on the current state using a fixed function. This function classifies the likely short-term effects of a given action (i.e. the likelihood of success or failure). Experimental results both in the RoboCup domain [Stone and Veloso, 1999b] and in the network packet routing domain [Stone, 2000] demonstrate that TPOT-RL can effectively learn multi-agent control policies with very few training examples in complex, dynamic domains. However, [Stone and Sutton, 2001] admit that such state aggregation approaches are not well suited to learning complex functions.

Perhaps the work most closely related to the one presented in this chapter is [Stone et al., 2001] and [Stone and Sutton, 2001], who introduced the *3v2 Possession* challenge.

[Stone et al., 2001] experimented both with policy evaluation and policy learning. Policy evaluation was done using an omniscient agent that attempts to estimate the value of any given state from the attackers' perspective. Using these estimates as targets, they applied a supervised learning approach (Monte Carlo) to learning an approximation to the value function. In addition, they applied temporal-difference methods (TD(0)). Policy learning was done using Q-learning with each forward learning independently of the others. Following these experiments [Stone and Sutton, 2001] applied the Sarsa(λ) algorithm and replacing eligibility traces. One of the main differences with the work presented in this chapter is that [Stone et al., 2001] and [Stone and Sutton, 2001] used CMACs as a function approximator instead of KaBaGe. Another difference is the terminating condition for each episode. In the experiments presented in this chapter, instantaneous possession (i.e. for 1 cycle) by the defenders is enough to end an episode. However, [Stone et al., 2001] and [Stone and Sutton, 2001] used a much softer termination condition that requires the defenders to maintain possession for a limited period of time.

Chapter 7

Conclusions and Future Directions

7.1 Overview

This thesis contributes several techniques for developing successful agents to operate in real-time, noisy, collaborative, and adversarial environments. The aim of this chapter is to review the main scientific contributions to the fields of multi-agent systems (MAS) and machine learning (ML), and also outline some of the promising directions for future work. Although this work has been primarily motivated by the challenges inherent in the RoboCup domain, all contributions, apart from the Essex Wizards team, can be applied to domains with similar characteristics.

Chapter 2 described the main challenges of the RoboCup domain together with the key ideas behind the Robot World Cup Initiative. A detailed description of the server was also given. The characteristics of the server make it a challenging and realistic testbed for MAS research. Note that although the server provides with a communication model, this thesis contributes techniques that require no use of communication between teammates.

Chapter 3 introduced the functional requirements specification (FRS) for agents wishing to operate in the RoboCup and similar domains. A description of the architecture was given, including the components that make up the agent and the ways these components work together. In addition, the multi-threaded framework for real-time

agents was presented. Finally, the multi-threaded mediator pattern, a key component within the agent, was described together with the benefits of applying design patterns for the development of MAS.

Chapter 4 demonstrated how Kanerva's sparse distributed memory (SDM) model could be used as a generalisation component for tasks that involve large state spaces. This was a necessary first step towards the development of the KaBaGe-RL decision-making module, which was presented in chapter 6. To the best of the author's knowledge, there have been no attempts to combine Kanerva's SDM together with reinforcement learning (RL) methods.

Chapter 5 demonstrated how RL methods (in particular Q-learning) could be used as a decision-making mechanism in the RoboCup and similar domains. In addition, the simple *2 Attackers versus 1 Defender* challenge has demonstrated the ability of an agent to exhibit co-operative behaviour via its decision-making module, even when it has no knowledge of the concept of co-operation. This was the second and final step towards the development KaBaGe-RL.

Finally, chapter 6 demonstrated how KaBaGe and RL (in particular residual Q-learning) could be combined in order to produce a decision-making module for noisy, real-time, collaborative and adversarial domains that involve large state spaces. The *3v2 Possession Football* challenge demonstrated the ability of an agent to exhibit co-operative behaviour via its decision-making module even when it has no knowledge of the concept of co-operation. Experimental results showed that KaBaGe-RL performs significantly better than the benchmark policies used.

7.2 Contributions

The four main contributions of this thesis are:

1. **A generalised agent architecture** suitable for complex *real-time AI* domains [Musliner et al., 1995] that require both deliberation and reactivity. The proposed model combines the standard *sense-think-act* cycle (seen in reactive architectures) with a *probabilistic world-model* (seen in deliberative architectures). In addition, the architecture provides a *multi-threaded mediator*, which simplifies the inter-communication between the various components within the agent.

2. **A multi-threaded implementation** of the agent's *sense-think-act* cycle for improved performance and bounded response times. The proposed implementation considers three threads that operate concurrently (or in parallel if multi-processor hardware is available). The proposed model performs significantly better than the existing single-threaded ones, given the asynchronous sensing and acting rates of the agent and the slow communication links between the agent and the environment.

3. **The KaBaGe-RL decision-making module**, which combines a generalisation module together with RL based action selection. The generalisation module is based on Kanerva's sparse distributed memory (SDM) and is responsible for expanding the original input representation received by the agent. The action selection module is based on residual Q-learning and is responsible for both choosing an action and updating the weights of the function approximator.

4. **The Essex Wizards Robotic Soccer System**, which is a fully implemented and operational team of simulated robotic soccer agents. The documented description of this research, together with the algorithmic details and the source code, should enable future researchers to build upon the successful implementation of the Essex Wizards team.

7.3 Future Directions

The research reported in this thesis has been used in the development of the Essex Wizards team that participated in a number of RoboCup competitions. The first such competition was the third international RoboCup championship (an event with 36 teams competing in the simulation league), held in Stockholm, during 1999. The Wizards qualified as the first team in their round robin group and went on to achieve a combined score of 59-9 over their opponents, finishing in third place. The following year, during the first European RoboCup championship (an event with 14 teams competing in the simulation league), the Wizards qualified as the second team in their round robin group and went on to achieve a combined score of 53-4 over their opponents, finishing in third place. While the Essex Wizards performed successfully in competitions as a result of this research, there are more challenges within the robotic soccer domain (and domains with similar characteristics) to be addressed. This section highlights some of these challenges and also comments on future research directions towards the contributed algorithms.

7.3.1 Future Directions within RoboCup

- Opponent modelling

  As described in chapter 2, opponent modelling is one of the major challenges in the RoboCup domain. Reasoning about other agents' intentions can be very beneficial in adversarial environments. Throughout this thesis, opponents were either ignored or treated as part of the environment. Recent additions to the server, such as the on-line coach, allow for strategy recognition and adaptation of a team's behaviour in order to exploit weaknesses in the opposition. Apart from the coach, individual agents can also benefit from predicting opponents' intentions, especially if the opponents use fixed strategies (e.g. for passing).

- Heterogeneous agents and emergent co-operation

  The latest version of the server introduced the ability to form a team using heterogeneous agents. Until recently only the goalkeeper differed from other agents in that it had the ability to catch the ball. However, it is now possible to construct a team with agents that have different properties. For example there are agents that can run faster but get tired quicker. Another example is agents that can kick the ball harder but are less accurate. It would be interesting to investigate the benefits of emergent co-operation within RoboCup, in a heterogeneous setting. Would it be possible for a team to learn to fully utilise the benefits of the different attributes each team member has to offer? How would the formation of the team change in order to

accommodate all the different properties of each agent? What would the end result look like?

- Scaling towards 11 versus 11 games

Chapters 5 and 6 demonstrated that moving from a trivial 2v1 (i.e. a total of 3 agents) co-operative scenario to a 3v2 (i.e. a total of 5 agents) one significantly increases the complexity of the learning task. A long-term research goal is to scale up towards 11v11 games. One of the main challenges is for agents to be able learn quickly enough since a game lasts for only 6000 simulated cycles. Can an effective policy be learned during this short interval? Will there be enough time to apply this policy and overcome the penalties that were imposed during the initial stages of the training?

7.3.2 Future Directions within the Contributed Algorithms

- Use unsupervised learning within KaBaGe to discard inefficient prototypes

The experiments in chapters 4 and 6 used randomly constructed prototypes for the KaBaGe module. Although this method provided with good results, it is potentially inefficient. Depending on the input distribution, it is possible that some prototypes may always be active while others may always remain inactive. As described in [Sutton and Whitehead, 1993] such units do not contribute to learning because they cannot discriminate between inputs. However, it should be possible to use some form of unsupervised learning in order to keep the frequency and density of activation within a desirable range.

- Apply KaBaGe-RL to other domains to evaluate the generality of the approach

  The tested for most of the experiments in this thesis has been the RoboCup domain. A future research goal is to evaluate the generality of KaBaGe-RL by applying it in other domains. A potential candidate is the network packet routing domain [Boyan and Littman, 1994], which [Stone, 2000] used in order to evaluate the generality of TPOT-RL (as described in chapter 6). The network packet routing problem can be seen as a multi-agent collaborative task, where an independent agent represents each node in the network. When a packet arrives at a given node the agent has to decide to which neighbouring node to forward it. The team's goal is to find an efficient way to distribute packets across the network. Although there are no adversaries involved in this setting, it would be interesting to test the performance of KaBaGe-RL on this collaborative task.

- Compare the efficiency of KaBaGe-RL with other decision-making modules

  KaBaGe-RL is a decision-making module that combines a generalisation module based on Kanerva's SDM together with residual Q-learning. It would be interesting to investigate combinations of other function approximation and action selection methods. A number of examples already exist (such as CMACs and Q-learning [Stone et al., 2001] or CMACs and Sarsa [Stone and Sutton, 2001]) but there are still a number of interesting candidates both for generalisation and action selection components. How will principal components analysis (PCA) perform comparing to KaBaGe? How will KaBaGe work when combined with Sarsa?

7.4 Concluding Remarks

In recent years, two major areas of computer science have started converging. AI research is moving towards realistic domains requiring real-time responses, and real-time systems are moving towards more complex applications requiring intelligent behaviour. This thesis contributes techniques for building agents in a particularly complex class of real-world domains that are noisy, require real-time decision-making, and are both collaborative and adversarial. It is hoped that the contributions of this thesis will prove useful in developing applications for similar domains and will enable future researchers move closer to the creation of complete AI agents.

Appendix A

List of Acronyms

Following is the list of all acronyms that were used throughout this thesis.

ADN: Address Decoding Neuron

AI: Artificial Intelligence

API: Application Programming Interface

CMAC: Cerebellum Model Articulation Controller

DP: Dynamic Programming

ER: Expanded Representation

FRS: Functional Requirements Specification

I/O: Input/Output

IPD: Iterated Prisoner's Dilemma

KaBaGe: Kanerva Based Generalisation

LMS: Least Mean Square

LTU: Linear Threshold Unit

MAMDP: Multi-Agent Markov Decision Process

MAS: Multi-Agent System

MDP: Markov Decision Process

ML: Machine Learning

OCM: Observing Co-ordination Mechanism

OO: Object Oriented

PCA: Principal Component Analysis

PCM: Perceptual Co-ordination Mechanism

RBF: Radial Basis Functions

RL: Reinforcement Learning

RR: Random Representation

SDM: Sparse Distributed Memory

TCP/IP: Transmission Control Protocol/Internet Protocol

TD: Temporal Difference

TPOT-RL: Team-Partitioned Opaque-Transition Reinforcement Learning

UDP/IP: User Datagram Protocol/Internet Protocol

Appendix B

Source Code

The sources of the Essex Wizards team are an on-line appendix. The whole of the 1999 version was released under the terms of the GNU General Public License (version 2). The appendix also contains binaries (for linux) of recent competition-ready editions of the team (e.g. Euro-RoboCup 2000 and Melbourne 2000) and various software tools that were used during development. A direct link is not given here, as it may soon be outdated. The best way to access these pages is through the *University of Essex* homepage located at:

http://www.essex.ac.uk/

From there follow the links to the *Computer Science* department and then the *Robotics and Intelligent Machines Research Group*.

Appendix C

Server Parameters

The following table lists some operational parameters used by the server together with their default values and a brief description.

Table C.1: Server parameters.

Parameter Name	Default Value	Description
field_width	68	Width of the field
field_length	105	Length of the field
feel_distance	3	Objects within 3 meters appear in the visual string even if they are not within the view cone
unum_far_length	20	Both uniform number and team name are visible if the sensed agent is less than 20 meters away
unum_too_far_length	40	The uniform number is not visible if the sensed agent is further than 40 meters away
team_far_length	40	Team name is visible if the sensed agent is less than 40 meters away
team_too_far_length	60	Uniform number and team name are not visible if the sensed agent is further than 60 meters away
goal_width	14.02	Width of the goal
player_size	0.3	Radius of an agent
player_decay	0.4	Decay rate of agent's speed
player_rand	0.1	Amount of noise added in agents movements and turns
player_speed_max	1	Max speed an agent can reach
stamina_max	3500	Maximum stamina of an agent
stamina_inc_max	35	Amount of stamina an agent gains in a cycle
recover_dec_thr	0.3	Decrement threshold for an agent's recovery
recover_dec	0.002	Decrement step for an agent's recovery
recover_min	0.5	Minimum agent recovery
effort_dec_thr	0.3	Decrement threshold for an agent's effort
effort_dec	0.005	Decrement step for an agent's effort
effort_inc_thr	0.6	Increment threshold for an agent's effort
effort_inc	0.01	Increment step for an agent's effort
effort_min	0.6	Minimum value for an agent's effort
hear_max	2	Maximum hearing capacity of an agent

hear_inc	1	Minimum hearing capacity of an agent
hear_decay	2	Decay of hearing capacity of an agent
inertia_moment	5	Inertia moment of an agent
sense_body_step	100	Length of the interval for sending physical information (msec)
catchable_area_l	2	Goalkeeper's catchable area length
catchable_area_w	1	Goalkeeper's catchable area width
catch_probability	1	The probability for a goalkeeper to catch the ball
catch_ban_cycle	5	Minimum number of cycles between consecutive catch commands
ball_size	0.085	Radius of the ball
ball_decay	0.94	Decay rate of the ball speed
ball_rand	0.05	Amount of noise added in the movement of the ball
ball_speed_max	2.7	Maximum speed the ball can reach
kickable_margin	0.7	Used to calculate the radius of the kickable area
dash_power_rate	0.006	Used to calculate the power of dash requests sent to the server
kick_power_rate	0.016	Used to calculate the power of kick requests sent to the server
visible_angle	90	Angle of view cone of the agent during the default view mode (normal)
audio_cut_dist	50	Maximum distance a message said by an agent can reach
quantize_step	0.1	Used for quantisation of distances of the ball of other agents
quantize_step_l	0.01	Used for quantisation of distances of the flags or lines
maxpower	100	Maximum allowed power
minpower	−100	Minimum allowed power
maxmoment	180	Maximum allowed moment
minmoment	−180	Minimum allowed moment
maxneckmoment	180	Maximum allowed neck moment
minneckmoment	−180	Minimum allowed neck moment
maxneckang	90	Maximum neck angle
minneckang	−90	Minimum neck angle
simulator_step	100	Duration (in msec) of each simulation cycle
send_step	150	Length of the interval for sending visual information (msec)
say_msg_size	512	Maximum size for audio messages (bytes)

Appendix D

References

[Abul et al., 2000] A. Abul, F. Polat, and R. Alhajj, *Multi-agent reinforcement learning using function approximation*, In IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and reviews, 30(4): 485-497, 2000.

[Agre and Chapman, 1990] P. E. Agre and D. Chapman, *What are plans for?*, In P. Maes, editor, Designing Autonomous Agents: Theory and Practise from Biology to Engineering and Back, MIT/Elsevier, 6:17-34, 1990.

[Albus, 1981] J. S. Albus, *Brains, Behaviour, and Robotics*, Byte Books, Subsidiary of McGraw-Hill, New Hampshire, 1981.

[Altenburg, 1994] K. Altenburg, *Adaptive Resource Allocation for a Multiple Mobile Robot System using Communication*, Technical Report NDSU-CSOR-TR-9404, North Dakota State University, 1994.

[Anderson, 1995] J. A. Anderson, *An Introduction to Neural Networks*, MIT Press, MA, 1995.

[Andou, 1998] T. Andou, *Refinement of soccer agents' positions using reinforcement learning*, In H. Kitano, editor, RoboCup-97: Robot Soccer World Cup I, Lecture Notes in Artificial Intelligence, Springer-Verlag, Berlin, pp. 373-388, 1998.

[Baird, 1995] L. C. Baird, *Residual algorithms: Reinforcement learning with function approximation*, In Proceedings of the twelfth International Conference on Machine Learning (ICML-95), Morgan-Kaufmann, San Francisco, 1995.

[Barman et al., 1993] R. A. Barman, J. J. Kingdon, A. K. Mackworth, D. K. Pai, M. K. Sahota, H. Wilkinson, and Y. Zhang, *Dynamite: A Testbed for Multiple Mobile Robots*, In Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-93), Workshop on Dynamically Interacting Robots, France, pp. 38-44, 1993.

[Barto, 1995] A. G. Barto, *Reinforcement Learning*, In M. A. Arbib, editor, Handbook of Brain Theory and Neural Networks, MIT Press, MA, 1995.

[Barto et al., 1983] A. G. Barto, R. S. Sutton, and C. W. Anderson, *Neuronlike elements that can solve difficult learning control problems*, In IEEE Transactions on Systems, Man, and Cybernetics, 13(5):835-846, 1983.

[Beckers et al., 1994] R. Beckers, O. E. Holland, and J. L. Deneubourg, *From Local Actions to Global Tasks: Stigmergy in Collective Robotics*, In R. Brooks and P. Maes, editors, Artificial Life IV, MIT Press, MA, 1994.

[Bellman, 1957] R. E. Bellman, *Dynamic Programming*, Princeton University Press, Princeton, 1957.

[Berenji, 1991] H. R. Berenji, *Artificial neural networks and approximate reasoning for intelligent control in space*, In Proceedings of the American Control Conference, pp. 1075-1080, 1991.

[Bertsekas, 1995] D. P. Bertsekas, *Dynamic Programming and Optimal Control*, Athena Scientific, MA, 1995.

[Bertsekas and Tsitsiklis, 1996] D. P. Bertsekas and J. N. Tsitsiklis, *Neuro-Dynamic Programming*, Athena Scientific, MA, 1996.

[Bottazzi et al., 1999] G. Bottazzi, G. Devetag, and G. Dosi, *Learning and Emergent Coordination in Speculative Markets: Some Properties of "Minority Game" Dynamics*, LEM--St. Anna's School of Advanced Studies, working paper (third draft), 1999.

[Boyan and Littman, 1994] J. A. Boyan and M. L. Littman, *Packet routing in dynamically changing networks: A reinforcement learning approach*, In J. D. Cowan, G. Tesauro, and J. Alspector, editors, Advances In Neural Information Processing Systems, Morgan-Kaufmann, San Francisco, 6:49-167, 1994.

[Boyan and Moore, 1995] J. A. Boyan and A. W. Moore, *Generalisation in reinforcement learning: Safely approximating the value function*, In G. Tesauro, D. S. Touretzky, and T. Leen, editors, In Proceedings of the Advances in Neural Information Processing Systems Conference, MIT Press, MA, 7:166, 1995.

[Boyan and Moore, 1996] J. A. Boyan and A. W. Moore, *Learning evaluation functions for large acyclic domains*, In L. Saitta, editor, In Proceedings of the thirteenth International Conference on Machine Learning (ICML-96), Morgan-Kaufmann, San Francisco, pp. 63-70, 1996.

[Boyan et al., 1995] J. A. Boyan, A. W. Moore, and R. S. Sutton, editors, *Proceedings of the Workshop on Value Function Approximation*, International Conference on Machine Learning, Technical Report CMU-CS-95-206, School of Computer Science, Carnegie Mellon University, PA, 1995.

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

[Broomhead and Lowe, 1988] D. S. Broomhead and D. Lowe, *Multi-variable functional interpolation and adaptive networks*, Complex Systems, 2:321-355, 1988.

[Burkhard et al., 1999] H. Burkhard, J. Wendler, T. Meinert, H. Myritz, and G. Sander, *AT Humboldt in RoboCup-99*, In M. Veloso, E. Pagello, and H. Kitano, editors, RoboCup-99: Robot Soccer World Cup III, Lecture Notes in Artificial Intelligence, Springer-Verlag, Berlin, 1856:542-545, 2000.

[Burmeister and Sundermeyer, 1992] B. Burmeister and K. Sundermeyer, *Cooperative problem-solving guided by intentions and perception*, In E. Werner and Y. Demazeau, editors, In Proceedings of the third European Workshop on Modelling Autonomous Agents in a Multi-Agent World, Kaiserlautern, Germany, pp. 77-92, 1992.

[Burns and Wellings, 1997] A. Burns and A. Wellings, *Real-time Systems and Programming Languages*, Addison-Wesley, Harlow, 1997.

[Bussmann and Demezeau, 1994] S. Bussmann and Y. Demazeau, *An agent model combining reactive and cognitive capabilities,* In Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS-94), Munchen, 1994.

[Butenhof, 1997] R.D. Butenhof, *Programming with POSIX Threads*, Addison-Wesley, Harlow, 1997.

[Ch'ng and Padgham, 1997a] S. Ch'ng and L. Padgham, *From Roles to Teamwork: A Framework and Architecture*, In Journal of Applied Artificial Intelligence, 12:211-232, 1997.

[Ch'ng and Padgham, 1997b] S. Ch'ng and L. Padgham, *Team description: Building Teams Using Roles, Responsibilities, and Strategies*, In Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-97): Workshop on RoboCup, 1997.

[Challet and Zhang, 1997] D. Challet and Y. C. Zhang, *Emergence of cooperation and organisation in an evolutionary game*, Physica, A 226:407, 1997.

[Challet and Zhang, 1998] D. Challet and Y. C. Zhang, *On the minority Game: Analytical and Numerical Studies*, Physica, A 256:514, 1998.

[Chen et al., 2001] M. Chen, E. Foroughi, F. Heintz, Z. Huang, S. Kapetanakis, K. Kostiadis, J. Kummeneje, I. Noda, O. Obst, P. Riley, T. Steffens, Y. Wang, and X. Yin, *Soccerserver manual version 7*, 2001.

[Claus and Boutilier, 1998] C. Claus and C. Boutilier, *The dynamics of reinforcement learning in cooperative multi-agent systems*, In Proceedings of the American Association for Artificial Intelligence (AAAI-98), AAAI Press, CA, 1998.

[Cohen et al., 1998] P. R. Cohen, H. J. Levesque, and I. Smith, *On team formation*, In J. Hintikka and R. Tuomela, editors, Contemporary Action Theory, Synthese, pp. 53, 1998.

[Coplien, 1992] J. O. Coplien, *Advanced C++ Programming Styles and Idioms*, Addison-Wesley, MA, 1992.

[Coradeschi and Karlsson, 1998] S. Coradeschi and L. Karlsson, *A Role-Based Decision-Mechanism for Teams of Reactive and Coordinating Agents*, In RoboCup-97: Robot Soccer World Cup I, Lecture Notes in Artificial Intelligence, Springer-Verlag, Berlin, pp. 112-122, 1998.

[Coulouris et al., 1995] G. Coulouris, J. Dollimore, and T. Kindberg, *Distributed Systems: Concepts and Design*, Addison-Wesley, Essex, 1995.

[Crites and Barto, 1996] R. H. Crites and A. G. Barto, *Improving Elevator Performance Using Reinforcement Learning*, In D. S. Touretzky, M. C. Mozer, and M. E. Hasselmo, editors, Advances in Neural Information Processing Systems, MIT Press, MA, 8:166, 1996.

[Dean et al., 1992] T. Dean, K. Basye, and J. Shewchuk, *Reinforcement learning for planning and control*, In S. Minton, editor, Machine Learning Methods for Planning and Scheduling, Morgan-Kaufmann, San Francisco, pp. 24-29, 1992.

[Doran et al., 1997] J. E. Doran, S. Franklin, N. R. Jennings, and T. J. Norman, *On cooperation in multi-agent systems*, The Knowledge Engineering Review, 12(3):309-314, 1997.

[Fasciano, 1995] M. J. Fasciano, *Real-time case-based reasoning in a complex world*, Technical Report TR-9605, University of Chicago, 1995.

[Fayad and Schmidt, 1997] M. E. Fayad and D. C. Schmidt, *Object-Oriented Application Frameworks*, Communications of the ACM, 40(10):32-38, 1997.

[Ferguson, 1992] I. A. Ferguson, *Touring Machines: An Architecture for Dynamic, Rational, Mobile Agents*, PhD thesis, Computer Laboratory, University of Cambridge, UK, 1992.

[Frankling and Graesser, 1997] S. Franklin and A. Graesser, *Is it an agent, or just a program?: A taxonomy for autonomous agents*, In J. P. Muller, M. J. Wooldridge, and N. R. Jennings, editors, Intelligent Agents III, Lectures Notes in Artificial Intelligence, Springer-Verlag, Heidelberg, 1193:21-36, 1997.

[French, 1991] R. M. French, *Using semi-distributed representations to overcome catastrophic forgetting in connectionist networks*, In Proceedings of the thirteenth Annual Cognitive Science Society Conference, NJ, pp. 173-178, 1991.

[Fudenberg and Levine, 1993] D. Fudenberg and D. K. Levine, *Steady state learning and Nash equilibrium*, Econometrica, 61(3):547-573, 1993.

[Gallant, 1993] S. I. Gallant, *Neural Network Learning and Expert Systems*, MIT Press, MA, 1993.

[Gallant and Smith, 1987] S. I. Gallant and D. Smith, *Random cells: an idea whose time has come and gone... and come again?*, In Proceedings of the IEEE International Conference on Neural Networks, 2:671-678, 1987.

[Gamma et al., 1995] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object Oriented Software*, Addison-Wesley, MA, 1995.

[Garvey and Lesser, 1993] A. Garvey and V. Lesser, *A survey of research in deliberative real-time artificial intelligence*, Real-Time Systems, 6:317-347, 1994.

[Georgeff and Ingrand, 1989] M. P. Georgeff and F. F. Ingrand, *Decision-making in embedded reasoning systems*, In Proceedings of the sixth International Joint Conference on Artificial Intelligence (IJCAI-89), Morgan-Kaufmann, San Mateo, pp. 972-978, 1989.

[Gordon, 1995] G. J. Gordon, *Stable function approximation in dynamic programming*, In A. Prieditis and S. Russell, editors, In Proceedings of the twelfth International Conference on Machine Learning (ICML-95), Morgan-Kaufmann, San Francisco, pp. 261-268, 1995.

[Grosz, 1996] B. J. Grosz, *Collaborative systems*, AI Magazine, 17(2):67-85, 1996.

[Guessoum and Dojat, 1996] Z. Guessoum and M. Dojat, *A real-time agent model in an asynchronous object environment*, In Proceedings of the seventh European Workshop on Modelling Autonomous Agents in a Multi Agent World (MAAMAW-96), NL, Springer-Verlag, Berlin, pp. 190-203, 1996.

[Hayes-Roth, 1990] B. Hayes-Roth, *Architectural foundations for real-time performance in intelligent agents*, Journal of Real-Time Systems, 2(1/2):99-125, 1990.

[Hillis, 1985] W. Hillis, *The Connection Machine*, MIT Press, MA, 1985.

[Hinton, 1984] G. E. Hinton, *Distributed Representations*, Technical Report CMU-CS-84-157, Department of Computer Science, Carnegie Mellon University, PA, 1984.

[Hinton, 1989] G. E. Hinton, *Connectionist learning procedures*, Artificial Intelligence, 40: 185-234, 1989.

[Horling, 1998] B. Horling, *JAF: A Reusable Component Architecture for Agent Construction*, MSc dissertation, University of Massachusetts, MA, 1998.

[Horvitz and Rutledge, 1991] E. J. Horvitz and G. Rutledge, *Time-dependent utility and action under uncertainty*, In Proceedings of the seventh Conference on Uncertainty in Artificial Intelligence, Los Angeles, pp. 151-158, 1991.

[Howe et al., 1990] A. E. Howe, D. M. Hart, and P. R. Cohen, *Addressing real-time constraints in the design of autonomous agents*, Journal of Real-Time Systems, 2(1/2):81-97, 1990.

[Hu and Wellman, 1998] J. Hu, and M. P. Wellman, *Multiagent reinforcement learning: Theoretical framework and an algorithm*, In J. Shavlik, editor, In Proceedings of the fifteenth International Conference on Machine Learning (ICML-98), Morgan-Kaufmann, NJ, pp. 242-250, 1998.

[Hu et al., 1999] H. Hu, K. Kostiadis, and Z. Liu, *Coordination and Learning in a Team of Mobile Robots*, In Proceedings of the IASTED Robotics and Applications Conference, CA, pp. 378-383, 1999.

[Hu et al., 2000] H. Hu, K. Kostiadis, M. Hunter, and M. Seabrook, *Essex Wizards '99 Team Description*, In M. Veloso, E. Pagello, and H. Kitano, editors, RoboCup-99: Robot Soccer World Cup III, Lecture Notes in Artificial Intelligence, Springer-Verlag, Berlin, 1856:558-562, 2000.

[Hu et al., 2001] H. Hu, K. Kostiadis, M. Hunter, and N. Kalyviotis, *Essex Wizards 2000 Team Description*, In P. Stone, T. Balch, and G. Kraetszchmar, editors, RoboCup-00: Robot Soccer World Cup IV, Lecture Notes in Artificial Intelligence, Springer-Verlag, Berlin, pp. 441-444, 2001.

[Hu et al., 2002] H. Hu, K. Kostiadis, M. Hunter, and N. Kalyviotis, *Essex Wizards 2001 Team Description*, In A. Birk, S. Coradeschi, and S. Tadokoro, editors, RoboCup-01: Robot Soccer World Cup V, Lecture Notes in Artificial Intelligence, Springer-Verlag, Berlin, 2002.

[Hunter, 2002] M. Hunter, *Building Position Selection Behaviours for Simulated Soccer Agents*, PhD thesis, Department of Computer Science, University of Essex, UK, 2002.

[Hunter et al., 2000] M. Hunter, K. Kostiadis, and H. Hu, *A Behaviour-based Approach to Position Selection for Simulated Soccer Agents*, In Proceedings of the first European RoboCup Workshop, Amsterdam, 2000.

[Jaakkola et al., 1995] T. Jaakkola, S. P. Singh, and M. I. Jordan, *Monte-Carlo reinforcement learning in non-Markovian decision problems*, In G. Tesauro, D. S. Touretzky, and T. K. Leen, editors, Advances in Neural Information Processing Systems, MA, 7:345-352, 1995.

[Jaeckel, 1989a] L. A. Jaeckel, *An alternative Design for a Sparse Distributed Memory*, Technical Report TR-89.28, Research Institute for Advanced Computer Science (RIACS), NASA Ames Research Center, 1989.

[Jaeckel, 1989b] L. A. Jaeckel, *A class of designs for a sparse distributed memory*, Technical Report TR-89.30, Research Institute for Advanced Computer Science (RIACS), NASA Ames Research Center, 1989.

[Jung et al., 1997] D. Jung, G. Cheng, and A. Zelinsky, *An Experiment in Realising Cooperation between Autonomous Mobile Robots*, In Proceedings of the fifth International Symposium on Experimental Robotics (ISER-97), Barcelona, 1997.

[Kaelbling, 1996] L. P. Kaelbling, editor, *Special issue of Machine Learning on reinforcement learning*, 22, 1996.

[Kaelbling et al., 1996] L. P. Kaelbling, M. L. Littman, and A. W. Moore, *Reinforcement Learning: A survey*, Journal of Artificial Intelligence Research, 4:237-285, 1996.

[Kalai and Lehrer, 1993] E. Kalai and E. Lehrer, *Rational learning leads to Nash equilibrium*, Econometrica, 61(5):1019-1045, 1993.

[Kanerva, 1988] P. Kanerva, *Sparse Distributed Memory*, MIT Press, MA, 1988.

[Karlsson, 1995] R. Karlsson, *Evaluation of a Fast Activation Mechanism for the Kanerva SDM Memory*, SICS Research Report R95:10, Swedish Institute of Computer Science, 1995.

[Keeler, 1988] J. D. Keeler, *Comparison between Kanerva's SDM and Hopfield-type Neural Networks*, Cognitive Science, 12:299-329, 1988.

[Keerthi and Ravindran, 1997] S. S. Keerthi and B. Ravindran, *Reinforcement Learning*, In E. Fiesler and R. Beale, editors, Handbook of Neural Computation, Oxford University Press, NY, chapter 3, 1997.

[Kitano et al., 1995] H. Kitano, M. Asada, Y. Kuniyoshi, I. Noda, and E. Osawa, *RoboCup: The Robot World Cup Initiative*, In Proceedings of the twelfth International Joint Conference on Artificial Intelligence (IJCAI-95), Workshop on Entertainment and AI/Artificial Life, 1995.

[Kitano et al., 1997] H. Kitano, M. Tambe, P. Stone, M. Veloso, S. Coradeschi, E. Osawa, H. Matsubara, I. Noda, and M. Asada, *The RoboCup synthetic agent challenge 97*, In Proceedings of the fourteenth International Joint Conference on Artificial Intelligence (IJCAI-97), Morgan-Kaufmann, San Francisco, pp. 24-29, 1997.

[Kohonen, 1990] T. Kohonen, *The self-organising map*, In Proceedings of the IEEE 78:1464-1480, 1990.

[Kostiadis and Hu, 1999] K. Kostiadis and H. Hu, *Reinforcement Learning and Co-operation in a Simulated Multi-agent System*, In Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS-99), Korea, pp. 990-995, 1999.

[Kostiadis and Hu, 2000] K. Kostiadis and H. Hu, *A Multi-threaded Approach to Simulated Soccer Agents for the RoboCup Competition*, In M. Veloso, E. Pagello, and H. Kitano, editors, RoboCup-99: Robot Soccer World Cup III, Lecture Notes in Artificial Intelligence, Springer-Verlag, Berlin, 1856:366-377, 2000.

[Kostiadis et al., 2000] K. Kostiadis, M. Hunter, and H. Hu, *The Use of Design Patterns for the Development of Multi-Agent Systems*, In Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics (SCM-00), Tennessee, 2000.

[Kostiadis and Hu, 2001] K. Kostiadis and H. Hu, *KaBaGe-RL: Kanerva Based Generalisation and Reinforcement Learning for Possession Football*, In Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS-01), Maui, pp. 292-297, 2001.

[Kristoferson, 1995] J. Kristoferson, *Some comments on the Information Stored in Sparse Distributed Memory*, SICS Research Report R95:11, Swedish Institute of Computer Science, 1995.

[Kube and Zhang, 1992] C. R. Kube and H. Zhang, *Collective Robotic Intelligence*, In Proceedings of the seconds International Workshop on Simulation of Adaptive Behaviour, pp. 460-468, 1992.

[Laird et al., 1987] J. E. Laird, A. Newell, and P. S. Rosenbloom, *SOAR: an architecture for general intelligence*, Artificial Intelligence, 33(1):1-62, 1987.

[Lauer and Riedmiller, 2000] M. Lauer and M. Riedmiller, *An algorithm for distributed reinforcement learning in cooperative multi-agent systems*, In Proceedings of the seventeenth International Conference on Machine Learning (ICML-00), CA, pp. 535-542, 2000.

[Lee, 1991] C. C. Lee, *A self learning rule-based controller employing approximate reasoning and neural net concepts*, International Journal of Intelligent Systems, 6(1):71-93, 1991.

[Lin, 1992] L. J. Lin, *Self-improving reactive agents based on reinforcement learning, planning, and teaching*, Machine Learning, 8(3/4):293-322, 1992.

[Lin and Kim, 1991] C. S. Lin and H. Kim, *CMAC-based adaptive critic self-learning control*, In IEEE Transactions on Neural Networks, 2:530-533, 1991.

[Lippman and Stroustrup, 1988] S. Lippman and B. Stroustrup, *Pointers to Class Members in C++*. Proceedings of USENIX C++ Conference, Denver, pp. 305-323, 1988.

[Littman, 1994] M. L. Littman, *Markov games as a framework for multi-agent reinforcement learning*, In W. W. Cohen and H. Hirsh, editors, In Proceedings of the eleventh International Conference on Machine Learning (ICML-94), Morgan-Kauffman, NJ, pp. 157-163, 1994.

[Mataric, 1994] M. J. Mataric, *Interaction and intelligent behaviour*, PhD thesis AITR-1495, MIT EECS, MIT AI Lab, 1994.

[Mataric, 1995] M. J. Mataric, *Issues and approaches in the design of collective autonomous agents*, Journal of Robotics and Autonomous Systems, 16:321-331, 1995.

[Menczer and Monge, 1999] F. Menczer and A. E. Monge, *Scalable Web Search by Adaptive Online Agents: An InfoSpiders Case Study*, In M. Klusch, editor, Intelligent Information Agents, Springer-Verlag, Berlin, pp. 323-347, 1999.

[Miller et al., 1994] W. T. Miller, S. M. Scalera, and A. Kim, *Neural network control of dynamic balance for a biped walking robot*, In Proceedings of the eighth Yale Workshop on Adaptive and Learning Systems, Yale University, New Haven, pp. 156-161, 1994.

[Mitchell, 1997] T. M. Mitchell, *Machine Learning*, McGraw-Hill International Editions, Singapore, 1997.

[Moody, 1989] J. E. Moody, *Fast learning in multi-resolution hierarchies*, In D. Touretzky, editor, Advances in Neural Information Processing, Morgan-Kaufmann, Los Altos, 1:29-39, 1989.

[Moody and Darken, 1989] J. E. Moody and C. J. Darken, *Fast learning in networks of locally-tuned processing units*, Neural Computation 1:281-294, 1989.

[Muller, 1996] J. P. Muller, *The Design of Autonomous Agents – A Layered Approach*, Lecture Notes in Artificial Intelligence, Springer-Verlag, Berlin, 1177:101, 1996.

[Muller, 1998] J. P. Muller, *The right agent (architecture) to do the right thing*, In J.P. Muller, M. P. Singh, and A. S. Rao, editors, Intelligent Agents V – Proceedings of the

fifth International Workshop on Agent Theories, Architectures and Languages (ATAL98), Lecture Notes in Artificial Intelligence, Springer-Verlag, Heidelberg, 1555:211-225, 1998.

[Murray et al., 2001] J. Murray, O. Obst, and F. Stolzenburg, *RoboLog Koblenz 2000*, In P. Stone, T. Balch, and G. Kraetzschmar, editors, RoboCup-2000: Robot Soccer World Cup IV, Lecture Notes in Articial Intelligence, Springer-Verlag, Berlin, 2001.

[Musliner et al., 1993] D. J. Musliner, E. H. Durfee, and K. G. Shin, *CIRCA: A cooperative intelligent real-time control architecture*, IEEE Transactions of Systems, Man, and Cybernetics, 23(6):1561-1574, 1993.

[Musliner et al., 1995] D. J. Musliner, J. A. Hendler, A. K. Agrawala, E. H. Durfee, J. K. Strosnider, and C. J. Paul, *The Challenge of Real-Time Artificial Intelligence*, In IEEE Computer, 28(1):58--66, 1995.

[Nakashima et al., 1995] H. Nakashima, I. Noda, and I. Ohsawa, *Organic programming for multi-agents*, In V. Lesser, editor, In Proceedings of the first International Conference on Multi-agent Systems, San Francisco, (poster) pp. 459, 1995.

[Nissanke, 1997] N. Nissanke, *Realtime Systems,* Prentice Hall, London, 1997.

[Noda, 1998] I. Noda, *Team GAMMA: Agent programming on Gaea*, In H. Kitano, editor, RoboCup-97: Robot Soccer World Cup I, Lecture Notes in Artificial Intelligence, Springer-Verlag, Berlin, pp. 205, 1998.

[Noda et al., 1996] I. Noda, H. Matsubara, and K. Hiraki, *Learning Cooperative Behaviour in Multi-agent Environment: a case study of choice of play-plans in soccer*, In Proceedings of the Pacific Rim International Conference on Artificial Intelligence, Cairns, pp. 570-579, 1996.

[Noda et al., 1998] I. Noda, H. Matsubara, K. Hiraki, and I. Frank, *Soccer Server: a tool for research on multi-agent systems*, Applied Artificial Intelligence, 12:233-250, 1998.

[Noreils, 1993] F. R. Noreils, *Toward a Robot Architecture Integrating Cooperation between Mobile Robots: Application to Indoor Environment*, In the International Journal of Robotics Research, 12(1):79-98, 1993.

[Norman et al., 1997] T. J. Norman, N. R. Jennings, P. Faratin, and E. H. Mamdani, *Designing and implementing a multi-agent architecture for business process management*, In J. P. Muller, M. J. Wooldridge, and N. R. Jennings, editors, Intelligent Agents III, Lecture Notes in Artificial Intelligence, Springer-Verlag, Heidelberg, 1193:261-275, 1997.

[Nwana, 1996] H. S. Nwana, *Software Agents: an overview*, Knowledge Engineering Review, 11(3):205-244, 1996.

[Occello and Demazeau, 1994] M. Occello and Y. Demazeau, *Building real time agents using parallel blackboards and its use for mobile robotics*, In Proceedings of the IEEE International Conference on Systems, Man and Cybernetics (SMC-94), San Antonio, 1994.

[Pfeifer and Wainer, 2000] J. Pfeifer and J. Wainer, *Multi-Agent Plans: Analysis and Implementation*, In L. N. Barros, R. M. Cesar Jr., F. G. Cozman, and A. H. Reali Costa, editors, International Joint Conference (IBERAMIA-00) and (SBIA-00), Workshop Proceedings: Meeting on Multi-Agent Collaborative and Adversarial Perception, Planning, Execution, and Learning, Sao Paulo, 2000.

[Pomerleau, 1993] D. A. Pomerleau, *Knowledge-based training of artificial neural networks for autonomous robot driving*, In J. H. Connel and S. Mahadevan, editors, Robot Learning, Kluwer Academic Publishers, 1993.

[Prager and Fallside, 1988] R. W. Prager and F. Fallside, *The modified Kanerva model for automatic speech recognition*, Computer Speech and Language, 5:257-274, 1988.

[Rao and Ballard, 1995a] R. P. N. Rao and D. H. Ballard, *Object indexing using an iconic sparse distributed memory*, In Proceedings of the fifth International Conference on Computer Vision (ICCV-95), pp. 24-31, 1995.

[Rao and Ballard, 1995b] R. P. N. Rao and D. H. Ballard, *Natural basis functions and topographic memory for face recognition*, In Proceedings of the fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95), Montreal, 1995.

[Rao and Fuentes, 1996] R. P. N. Rao and O. Fuentes, *Learning navigational behaviours using a predictive sparse distributed memory*, In P. Maes, M. Mataric, J. A. Meyer, J. Pollack, and S. Wilson, editors, From Animals to Animats, MIT Press, MA, 4:382-390, 1996.

[Rao and Georgeff, 1995] A. S. Rao and M. P. Georgeff, *BDI-agents: from theory to practise*, In Proceedings of the first International Conference on Multi-agent Systems (ICMAS-95), San Francisco, pp. 312-319, 1995.

[Riedmiller, 2000] M. Riedmiller, *Concepts and facilities of a neural reinforcement learning control architecture for technical process control*, Journal of Neural Computing and Applications, 8:323-338, 2000.

[Riedmiller, et al., 2000] M. Riedmiller, S. Buck, A. Merke, R. Ehrmann, O. Thate, S. Dilger, A. Sinner, A. Hofmann, and L. Frommberger, *Karlsruhe Brainstormers – Design Principles*, In M. Veloso, E. Pagello, and H. Kitano, editors, RoboCup-99: Robot Soccer World Cup III, Lecture Notes in Artificial Intelligence, Springer-Verlag, Berlin, 1856:588-591, 2000.

[Riedmiller, et al., 2001] M. Riedmiller, A. Merke, D. Meier, A. Hoffmann, A. Sinner, O. Thate, and R. Ehrmann*, Karlsruhe brainstormers – a reinforcement learning approach to robotic soccer*, In P. Stone, T. Balch, and G. Kraetszchmar, editors, RoboCup-00: Robot Soccer World Cup IV, Lecture Notes in Artificial Intelligence, Springer-Verlag, Berlin, 2001.

[Riley, 1999] P. Riley, *Classifying Adversarial Behaviours in a Dynamic Inaccessible Multi-Agent Environment*, undergraduate senior thesis, Carnegie Mellon University, 1999.

[Rogers, 1989] D. Rogers, *Kanerva's Sparse Distributed Memory: An Associative Memory Algorithm Well-suited to the Connection Machine*, International Journal of High Speed Computing, 1(2):349-365, 1989.

[Rumelhart and McClelland, 1986] D. E. Rumelhart and J. L. McClelland, editors, *Parallel Distributed Processing: Explorations in the microstructures of cognition*, volume 1: Foundations, MIT Press, MA, 1986.

[Rumelhart et al., 1986] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, *Learning internal representations by error propagation*, In D. E. Rumelhart and J. L. McClelland, editors, Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Bradford/MIT Press, MA, 1:318-362, 1986.

[Rummery and Niranjan, 1994] G. A. Rummery and M. Niranjan, *On-line Q-learning using connectionist systems*, Technical Report CUED/F-INFENG/TR 166, Department of Engineering, Cambridge University, 1994.

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

[Sabes, 1993] P. Sabes, *Approximating Q-values with basis function representations*, In Proceedings of the fourth Connectionist Models Summer School, NJ, 1993.

[Samuels, 1959] A. L. Samuels, *Some studies in machine learning using the game of checkers*, IBM Journal of Research and Development, 3(3):210-229, 1959.

[Sandholm and Crites, 1996] T. W. Sandholm and R. H. Crites, *On Multiagent Q-Learning in a Semi-competitive Domain*, In G. Weiband and S. Sen, editors, Adaptation and Learning in Multiagent Systems, Springer-Verlag, Berlin, pp. 188, 1996.

[Schmidt, 1997] D. Schmidt, *Handbook of Programming Languages*, In P. Salus, editor, volume 1, MacMillan Computer Publishing, 1997.

[Sen et al., 1994] S. Sen, M. Sekaran, and J. Hale, *Learning to coordinate without sharing information*, In Proceedings of the American Association for Artificial Intelligence (AAAI-94), pp. 426-431, 1994.

[Shewchuk and Dean, 1990] J. Shewchuk and T. Dean, *Towards learning time-varying functions with high input dimensionality*, In Proceedings of the fifth IEEE International Symposium on Intelligent Control, IEEE Computer Society Press, CA, pp. 383-388, 1990.

[Sian, 1991] S. Sian, *Adaptation based on cooperative learning in multi-agent systems*, In Y. Demazeau and J. P. Muller, editors, Decentralise AI, Elsevier Science Publications, 2:257-272, 1991.

[Stevens, 1998] W.R. Stevens, *Unix Network Programming: Networking APIs: Sockets and XTI (volume 1)*, Prentice Hall International, London, 1998.

[Stolzenburg et al., 2000] F. Stolzenburg, O. Obst, J. Murray, and B. Bremer, *Spatial agents implemented in a logical expressible language*, In M. Veloso, E. Pagello, and H. Kitano, editors, RoboCup-99: Robot Soccer WorldCup III, Lecture Notes in Artificial Intelligence, Springer-Verlag, Berlin, 1856:481-494, 2000.

[Stone, 1998] P. Stone, *Layered Learning in Multi-Agent Systems*, PhD thesis, School of Computer Science, Carnegie Mellon University, 1998.

[Stone, 2000] P. Stone, *TPOT-RL Applied to Network Routing*, In Proceedings of the seventeenth International Conference on Machine Learning (ICML-00), CA, 2000.

[Stone and McAllester, 2001] P. Stone and D. McAllester, *An Architecture for Action Selection in Robotic Soccer*, In Proceedings of the fifth International Conference on Autonomous Agents (Agents-01), 2001.

[Stone and Sutton, 2001] P. Stone and R. S. Sutton, *Scaling Reinforcement Learning toward RoboCup Soccer*, In Proceedings of the eighteenth International Conference on Machine Learning (ICML-01), MA, pp. 206-212, 2001.

[Stone and Veloso, 1998] P. Stone and M. Veloso, *Communication in domains with unreliable, single-channel, low-bandwidth, communication*, In A. Drogoul, M. Tambe, and T. Fukuda, editors, Collective Robotics, Springer-Verlag, Berlin, pp. 54, 1998.

[Stone and Veloso, 1999a] P. Stone and M. Veloso, *Task Decomposition, Dynamic Role Assignment, and Low-Bandwidth Communication for Real-Time Strategic Teamwork*, Artificial Intelligence, 110(2):241-273, 1999.

[Stone and Veloso, 1999b] P. Stone and M. Veloso, *Team-Partitioned, Opaque-Transition Reinforcement Learning*, In M. Asada and H. Kitano, editors, RoboCup-98: Robot Soccer World Cup II, Lecture Notes in Artificial Intelligence, Springer-Verlag, Berlin, pp. 261-272, 1999.

[Stone et al., 1999] P. Stone, M. Veloso, and P. Riley, *The CMUnited-98 champion simulator team*, In M. Asada and H. Kitano, editors, RoboCup-98: Robot Soccer World Cup II, Lecture Notes in Artificial Intelligence, Springer-Verlag, Berlin, pp. 217, 1999.

[Stone et al., 2001] P. Stone, R. S. Sutton, and S. Singh, *Reinforcement Learning for 3 vs. 2 Keepaway*, In P. Stone, T. Balch, and G. Kraetszchmar, editors, RoboCup-2000: Robot Soccer World Cup IV, Lecture Notes in Artificial Intelligence, Springer-Verlag, Berlin, 2001.

[Sutton, 1984] R. S. Sutton, *Temporal Credit Assignment in Reinforcement Learning*, PhD thesis, University of Massachusetts, Amherst, 1984.

[Sutton, 1988] R. S. Sutton, *Learning to predict by the method of temporal differences, Machine Learning*, 3:9-44, 1988.

[Sutton, 1992] R. S. Sutton, editor, *Special issue of Machine Learning on reinforcement learning*, 8, 1992.

[Sutton, 1996] R. S. Sutton, *Generalisation in reinforcement learning: Successful examples using sparse coarse coding*, In D. S. Touretzky, M. C. Mozer, and M. E.

Hasselmo, editors, In Proceedings of the Advances in Neural Information Processing Systems Conference, MIT Press, MA, pp. 1038-1044, 1996.

[Sutton and Barto, 1998] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, MIT Press, MA, 1998.

[Sutton and Whitehead, 1993] R. S. Sutton and D. S. Whitehead, *Online learning with random representations*, In Proceedings of the tenth International Conference on Machine Learning, Morgan-Kaufmann, San Francisco, pp. 314-321, 1993.

[Tambe, 1997] M. Tambe, *Towards flexible teamwork*, Journal of Artificial Intelligence Research, 7:81-124, 1997.

[Tan, 1993] M. Tan, *Multi-Agent Reinforcement Learning: Independent vs. Cooperative Agents*, In Proceedings of the tenth International Conference on Machine Learning (ICML-93), MA, pp. 330-337, 1993.

[Tesauro, 1990] G. J. Tesauro, *Neurogammon: a neural network backgammon program*, In Proceedings (III) of the International Joint Conference on Neural Networks (IJCNN-90), pp. 33-40, 1990.

[Tesauro, 1992] G. J. Tesauro, *Practical issues in temporal difference learning*, Machine Learning, 8(3/4):257-277, 1992.

[Tesauro, 1994] G. J. Tesauro, *TD-Gammon, a self-teaching backgammon program, achieves master-level play*, Neural Computing, 6(2):215-219, 1994.

[Tesauro, 1995] G. J. Tesauro, *Temporal Difference Learning and TD-Gammon*, Communications of the ACM, 38:58-68, 1995.

[Thrun and Schwartz, 1993] S. Thrun and A. Schwartz, *Issues in using function approximation for reinforcement learning*, In Proceedings of the fourth Connectionist Models Summer School, NJ, pp. 255-263, 1993.

[Turk and Gorz, 1995] A. Turk and G. Gorz, *Kanerva's Sparse Distributed Memory: An Object-Oriented Implementation on the Connection Machine*, In Proceedings of the fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95), Montreal, pp. 473-479, 1995.

[Uther and Veloso, 1997] W. Uther and M. Veloso, *Generalizing Adversarial Reinforcement Learning*, In American Association for Artificial Intelligence Fall Symposium on Model Directed Autonomous Systems, pp. 206, 1997.

[Vidal and Buhler, 2001] J. M. Vidal and P. Buhler, *A Generic Agent Architecture for Multi-agent Systems*, Technical Report, University of South Carolina, 2001.

[Waltz and Fu, 1965] M. D. Waltz and K. S. Fu, *A heuristic approach to reinforcement learning control systems*, IEEE Transactions on Automatic Control, 10:390-398, 1965.

[Watkins, 1989] C. J. C. H. Watkins, *Learning from Delayed Rewards*, PhD thesis, Cambridge University, UK, 1989.

[Watkins and Dayan, 1992] C. J. C. H Watkins and P. Dayan, *Q-learning*, Machine Learning, 8:279-292, 1992.

[Weib, 1993] G. Weib, *Learning to coordinate actions in multi-agent systems*, In Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-93), 1993.

[Williams and Baird, 1993] R. J. Williams and L. C. Baird, *Tight Performance Bounds on Greedy Policies Based on Imperfect Value Functions*, Technical Report NU-CCS-93-14, Northeastern University, 1993.

[Woodworth, 1938] R. S. Woodworth, *Experimental Psychology*, H. Hold and co., NY, 1938.

[Wooldridge and Jennings, 1995a] M. J. Wooldridge and N. R. Jennings, *Agent Theories, Architectures, and Languages: A Survey*, In M. J. Wooldridge and N. R. Jennings, editors,

Proceedings ECAI-94 Workshop on Agent Theories, Architectures and Languages, Springer-Verlag, Berlin, pp. 1-12, 1995.

[Wooldridge and Jennings, 1995b] M. J. Wooldridge and N. R. Jennings, *Intelligent agents: Theory and practice*, Knowledge Engineering Review, 10(2):115--152, 1995.

[Wooldridge et al., 2000] M. J. Wooldridge, N.R. Jennings, and D. Kinny, *The Gaia methodology for agent-oriented analysis and design*, In Journal of Autonomous Agents and Multi-Agent Systems, 3(3):285-312, 2000.

[Zboril, 1997] F. Zboril, *An Application of Sparse Distributed Memory*, In Proceedings of the nineteenth International Workshop on Advanced Simulation of Systems (ASIS-97), Krnov, pp. 127-132, 1997.

[Zhang and Dietterich, 1995] W. Zhang and T. G. Dietterich, *A reinforcement learning approach to job-shop scheduling*, In Proceedings of the twelfth International Joint Conference on Artificial Intelligence (IJCAI-95), Morgan-Kaufmann, San Francisco, pp. 1114-1120, 1995.

[Zilberstein, 1993] S. Zilberstein, *Operational rationality through compilation of anytime algorithms*, PhD thesis, Department of Computer Science, University of California, Berkeley, 1993.