



the  
Osnabrueck RoboCup Agents  
Project

Andreas G.Nie	Angelika Hönemann	Andres Pegam
Collin Rogowski	Leonhard Hennig	Marco Diedrich
Philipp Hügelmeier	Sean Buttinger	Timo Steffens

compiled on November 14, 2001

# Contents

<b>1 Overview</b>	<b>4</b>
1.1 The developers and the project . . . . .	4
1.2 About this document . . . . .	4
1.2.1 How this document is organized . . . . .	4
1.2.2 How to read this document . . . . .	5
1.3 ORCAs architecture . . . . .	6
1.3.1 General information . . . . .	6
1.3.2 Playtree . . . . .	6
1.3.3 SFLS . . . . .	6
<b>2 CMU</b>	<b>8</b>
2.1 Introduction . . . . .	8
2.2 What is the CMU Code ? . . . . .	8
2.3 What problems does the CMU Code solve ? . . . . .	8
2.4 Which problems remain to be solved ? . . . . .	9
2.5 Adapting the code to the current Server Version . . . . .	9
<b>3 Learning</b>	<b>11</b>
3.1 Motivation . . . . .	11
3.2 Problem Structure . . . . .	11
3.3 Potential Learning Methods . . . . .	12
3.4 Temporal Difference Learning . . . . .	13
3.5 Grow Support . . . . .	15
3.6 ROUT . . . . .	16
3.7 Results . . . . .	17
3.8 Further Work . . . . .	17
<b>4 Playtree</b>	<b>19</b>
4.1 Introduction . . . . .	19
4.2 Formations . . . . .	21
4.3 Strategy . . . . .	22
4.3.1 Offense . . . . .	23
4.3.2 Defense . . . . .	26
4.3.3 Other PlayModes . . . . .	27
4.3.4 Conclusion . . . . .	27
4.4 Goalkeeping . . . . .	28
4.4.1 Introduction . . . . .	28
4.4.2 Watching the ball . . . . .	29

4.4.3	Positioning and Movement . . . . .	29
4.4.4	Catching and handling the ball . . . . .	30
4.4.5	Evaluation of goalie behavior . . . . .	31
4.4.6	Conclusion . . . . .	31
4.5	Conclusion . . . . .	31
<b>5</b>	<b>Communication</b>	<b>32</b>
5.1	Introduction . . . . .	32
5.2	Sharing knowledge about the current state of the world . . . . .	32
5.2.1	The protocol and the compression . . . . .	32
5.2.2	The architecture and structure . . . . .	33
5.2.3	Updating the world model from the message . . . . .	34
5.3	Communicating a plan . . . . .	34
5.3.1	How to express a plan in a message . . . . .	34
5.3.2	Information loss and relaying . . . . .	34
5.4	Promises of a communication using SFL . . . . .	35
<b>6</b>	<b>Logfile Analyzer</b>	<b>36</b>
6.1	Purpose of the Logfile-Analyzer . . . . .	36
6.2	The Logfile Analyzer's Basis: TimeSlice . . . . .	36
6.2.1	The purpose of TimeSlice . . . . .	36
6.2.2	Using TimeSlice . . . . .	37
6.2.3	A tool which uses TimeSlice: readLog . . . . .	37
6.3	SOM . . . . .	37
6.4	FOIL . . . . .	38
6.4.1	What it is . . . . .	38
6.4.2	Why we chose it . . . . .	38
6.4.3	What we did with it . . . . .	38
<b>7</b>	<b>Online Coach</b>	<b>41</b>
7.1	Introduction . . . . .	41
7.2	The online coach in RoboCup . . . . .	41
7.3	The standard coach language . . . . .	41
7.4	The ORCA online coach . . . . .	43
7.4.1	General approach . . . . .	43
7.4.2	Marking . . . . .	43
7.4.3	Defensive formations . . . . .	44
7.4.4	Detecting opponent setplays and formations . . . . .	45
7.5	Experiences drawn from the first coach competition at RoboCup 2001 . . . . .	45
7.6	Conclusion . . . . .	46
<b>8</b>	<b>SFLS</b>	<b>47</b>
8.1	Introduction . . . . .	47
8.2	Strategy Formalization Language - Concepts . . . . .	47
8.2.1	Abstracting Clang concepts . . . . .	47
8.2.2	Control keywords . . . . .	48
8.2.3	Conditions . . . . .	49
8.2.4	Actions . . . . .	49
8.3	Implementing SFL . . . . .	49
8.3.1	The parser . . . . .	49

8.3.2	The matcher . . . . .	50
8.3.3	The selector . . . . .	51
8.3.4	The effector . . . . .	52
8.3.5	Integrating coach advice . . . . .	52
8.4	Conclusion . . . . .	52
<b>9</b>	<b>Testing, Debugging and Tuning</b>	<b>54</b>
9.1	The Gauntlet . . . . .	54
9.1.1	The purpose of a gauntlet . . . . .	54
9.1.2	Design criteria . . . . .	54
9.1.3	Implementation . . . . .	55
9.2	Quality Assurance Management . . . . .	56
9.2.1	Introduction . . . . .	56
9.2.2	Tasks and general principles . . . . .	57
9.2.3	Tools and Procedures . . . . .	57
9.2.4	Experiences with QAM . . . . .	57
9.2.5	Conclusion . . . . .	59
9.3	CVS . . . . .	59
<b>10</b>	<b>Tourneys</b>	<b>60</b>
10.1	RoboCup German Open 2001 in Paderborn . . . . .	60
10.2	RoboCup 2001 Seattle . . . . .	61
<b>11</b>	<b>Conclusion</b>	<b>63</b>
11.1	Achievements . . . . .	63
11.2	Acknowledgments . . . . .	63
<b>A</b>	<b>Debug-API</b>	<b>65</b>
A.1	Introduction . . . . .	65
A.2	Basics . . . . .	65
A.3	Examples . . . . .	66
A.4	Known Problems . . . . .	67
<b>B</b>	<b>Terms</b>	<b>68</b>
<b>C</b>	<b>SFLS Rule Writing</b>	<b>70</b>
C.1	General Concept . . . . .	70
C.2	Syntax . . . . .	70
C.3	Writing rules . . . . .	72
<b>D</b>	<b>SFL - grammar</b>	<b>74</b>
<b>E</b>	<b>netif.C</b>	<b>78</b>
<b>F</b>	<b>Authors</b>	<b>80</b>

# Chapter 1

## Overview

This document is about the RoboCup team developed by the Osnabrueck RoboCup (ORCA) student project at the University of Osnabrueck. Apart from discussing our concepts we will try to give advices regarding our programs and will give a report on our experience with the team.

### 1.1 The developers and the project

Being a student project at the University of Osnabrueck, Germany, our group consists of nine students, namely Andreas G. Nie, Andres Pegam, Angelika Hönemann, Collin Rogowski, Leonhard Hennig, Marco Diedrich, Philipp Hügelmeyer, Sean Buttinger, and Timo Steffens. Also we had two consultants on board: Prof. Dr. Claus Rollinger, and Wilfried Teiken.

We started out in October 2000 with a 1 year time period. During this time we participated in two tournaments: the German Championship in Paderborn, Germany and the World Championship in Seattle, USA. It might seem a little confusing but we used different names at those tournaments. During the German Open we participated as 'Osna BallByters' and finally we were known as 'Dirty Dozen'. Hence we refer to our team differently in parts of this document.

### 1.2 About this document

#### 1.2.1 How this document is organized

Following this chapter you may find:

- **CMU**  
the Carnegie Mellon University (CMU) released their code of their 1999 World Championship winning team so that other groups may use it and base their development on their basic client. Since we wanted to focus on strategy realization and online coaching we decided to use the CMU code as well. In this chapter we describe the changes we made.
- **Learning**  
While planing the development of our client it was decided early on that we want

to use learning methods to improve our team's skills. In this chapter are some words about the learning routines that we tried and why they didn't work for us.

- **Playtree**

Based on the CMU code we chose a decision tree concept to realize our strategy concepts. The specifics about this method are given in this chapter.

- **Communication**

Since the perceptive range of each client is limited communication is the means to keep every player on the field up to date. The chapter describes what is communicated and how it is done in our team.

- **Logfile Analyzer**

The SoccerServer produces logfiles for each game which can be viewed afterwards. There is a large library of past games which give an excellent way to analyze different teams. Our tool to do this is the logfile analyzer which is described in this chapter.

- **Online Coach**

Our online coach that participated in the 2001 Coach Competition in Seattle is introduced in this chapter. It explains how it works and how interaction with clients is realized.

- **SFLS**

When the playtree concept reached its limitations we had to come up with a new way of describing our strategies. In the process of doing so we developed the Strategy Formalization Language System (SFLS) which is presented in this chapter.

- **Testing, Debugging, and Tuning**

As always when a large group of programmers works on the same files there are bound to be conflicts. Some thoughts on the long road of developing and testing a functional team are given in this chapter.

- **Tourneys**

Finally, we share some of the experiences we gathered during the participation of the two tourneys in the last chapter.

Also we compiled an appendix in which we gathered some reference material mainly about SFLS.

## 1.2.2 How to read this document

This document was created as the final report of a one-year student project at the Institute of Cognitive Science, Osnabrueck. The project-guidelines demand that it must be possible to determine which project member wrote which sections. That is why there is an assignment from sections to names.

The chapters are pretty much self-contained. So you can skip chapters or read them in any order.

This document will give a detailed description of our team so that our general concepts may become clear. In combination with our source code it should be possible to try out own ideas and maybe base a different development on our work.

## 1.3 ORCAs architecture

### 1.3.1 General information

As we developed our client naturally we went through some changes. The most dramatic one is probably the introduction of SFLS (see Chapter 8) to our client. In the next chapters we discuss the different concepts thoroughly so right now we just give a broad overview on how the two approaches differ.

### 1.3.2 Playtree

The concept of a playtree is one that is probably used by most clients in the RoboCup Simulation League today. It is basically a decision tree which allowed us to construct a functional team in a short time. The team using the playtree participated in the German Open as well as the World Championship in Seattle and is discussed in more detail in Chapter 4.1

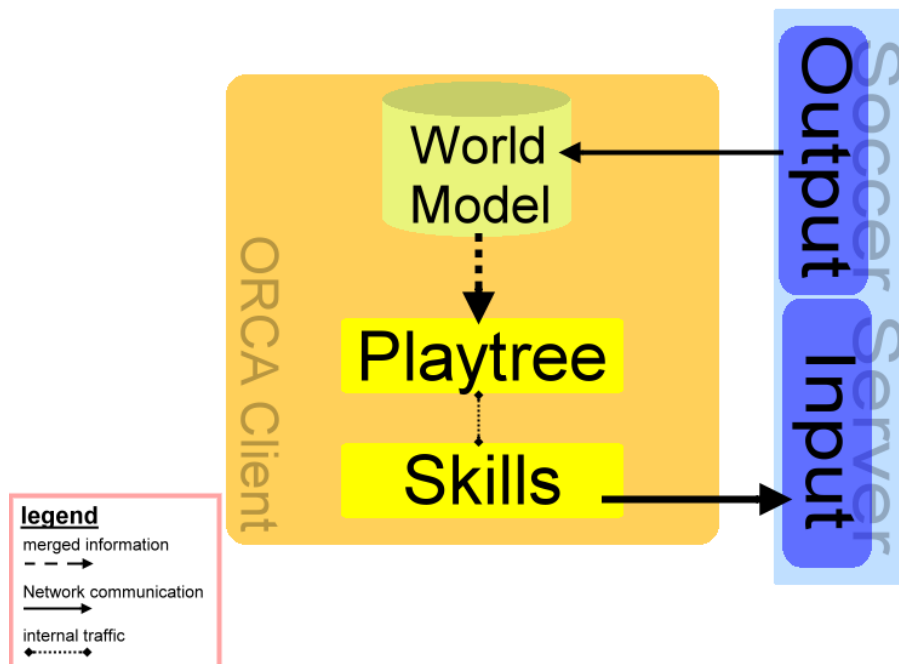


Figure 1.1: Playtree architecture

The flow of information in the playtree concept is rather simple: the messages coming from the SoccerServer are processed and stored in the World Model. Based on the current world state our playtree component decided on a certain action. It does so through a series of if-then constructs. Finally, the action is translated into a server conform format and send by the skill code module.

### 1.3.3 SFLS

With the development of a standard coach language (Clang) we discussed different approaches on how we could integrate those messages into our client. Through this

process we came up with the Strategy Formalization Language (SFL) which is an extension to Clang and which is realized in the SFL System described in chapter 8. Since the team that used SFLS understands the Clang it participated in the 2001 Online Coach Competition.

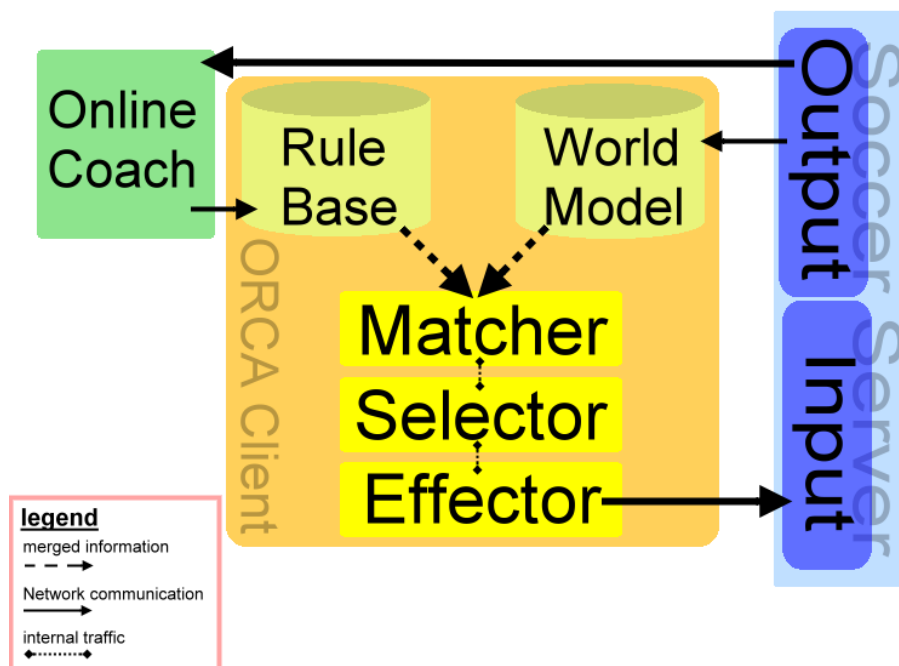


Figure 1.2: SFLS architecture

Different from the playtree approach the client has now two sources of input. On the one hand there are the direct messages from the server that are stored in the World Model. On the other hand are the (occasional) messages from our online coach. In fact, it doesn't even have to be our online coach, since our client works with any online coach. All the messages from the online coach (if available) are stored in the Rule Base. This Rule Base is already filled with the team's own strategic information. The online coach messages are therefore an addition to the given strategies and are supposed to improve the play of the team.

The crucial component of the SFLS architecture is the Matcher. It takes the current rules and matches them with the current World Model. Based on that match a rule is chosen through some heuristics by the Selector. When a rule is selected it is handed to the Effector. This module interprets the rule and converts it to the appropriate skill which is transferred to the server as the client's action for its current turn.

# Chapter 2

## CMU

### 2.1 Introduction

This section briefly describes why we use the CMU-Code[19]. You can get this code release from <http://www-2.cs.cmu.edu/afs/cs/usr/pstone/mosaic/RoboCup/CMUUnited99-sim.html>. It brings with it some basic features you can use with your agent implementation.

### 2.2 What is the CMU Code ?

It is the code that the team of the Carnegie Mellon University used in RoboCup 1999. Not really the whole code, but only the basic parts of it. It gives you the connectivity to the RoboCup server, a world model, that represents the world for the client, a basic time thread, basic actions like kick the ball, go to the ball etc. and some example functions for a very low level behavior.

You could say, that the CMU code is a framework that we used to get an easy start to RoboCup and to focus on the interesting parts of RoboCup. This is quite important, because normally you would have to put quite a lot of effort into caring about a consistent timing, client/server-communication and a correct update of the world model, which would keep you from focusing on the AI-parts in RoboCup.

### 2.3 What problems does the CMU Code solve ?

In the world model all information about the environment is stored. Every time cycle you get information about what you see, hear or sense with your body, you store this information to know where your opponents and your teammates are and where the ball is. You also get some internal information to calculate your position and your stamina. The problem is, that you get some of this information with noise, so you don't know the exact position of everything. Another feature the world model has is the ability to keep track of how accurate the information in the world model is.

The CMU world model provides a lot of functions to check the state of the world. For example there is a function to find an optimal position for intercepting the ball or to check whether it is possible to catch the ball. While in RoboCup you not only have to care about the actual world state, you also have to look into the future because the

world changes so fast, that you can't decide what to do on a fixed world model, you also have to make assumptions about the future world states. Part of this is already done in the CMU world model.

The communication with the server is also very important, so that you as a newbie to RoboCup don't have to care about timing problems in the client/server communication or missing any server messages.

The time thread takes care of sending the messages to the server and updating the world model at the given time. You can find more precise information about the steps taken during a world model update in a time cycle in the README file distributed with the code.

The basic actions are also quite important when you start with RoboCup, because they do not simply perform the action you want to do by putting together the basic actions, that you can send to the server (kick, turn, dash and catch for the goalie). It also calculates the correct value to be send and checks, whether these actions are possible.

It also contains some help functions to calculate geometric figures and positions on the field. Also there are some samples for very low level behavior to check out how to use the code.

## 2.4 Which problems remain to be solved ?

Although this "framework" solves a lot of tasks in RoboCup, quite a lot things still remain to be done. There is no communication between the agents, so you have to think about timing and a good plan for communication.

There is no high level planning, so you have to think about the behavior in different situations. You have to determine, whether you are in offense or in defense and how to behave in so called standard situations, meaning for example whether to go to the ball and kick it in a free kick situation, run free or do something else.

There are no classes for positioning and formation, which are quite important, because tactical behavior in a game (e.g. good positioning, playing with off-side calls etc.) is an important skill in playing good RoboCup soccer. You have to think about a situation-dependent positioning.

## 2.5 Adapting the code to the current Server Version

Since the release of the code some things were changed in the SoccerServer. You can find a list of all changes to the Soccer Server in the CHANGES file delivered with the Soccer-Server sources. Our client was adapted to the Soccer Server 7.10, so further changes are not in the code base. The major changes we had to care about are:

- Heterogenous players - Before each game, the server creates 6 different player types with different abilities in speed, kick range, kick power, preciseness of kick, stamina etc.. You have to find out the correct player type of each opponent and store each teammates player type to make assumptions about the future. Since there is a huge amount of information input, we do not use all information send to us by the server.
- Server Communication - There where some changes in server messages, that had to be parsed differently. There also some new messages, that have not been send

by earlier versions of the Soccer Server.

- The Standard Coach Language (Clang) - Since 2001 there is a Standard Coach Language, that can be used by Online Coaches.
- Time Threading - Even though not mentioned in the CHANGES file explicitly, following the heterogenous player messages, there were some changes in the communication between client and server.
- Parameter Values - Some default values changed and took an influence on the skills. For example, the optimal kick was totally changed. You have to kick only with full power in the right directing to get the optimal acceleration of the ball, while using earlier Server Versions you had to kick the ball around you to get an optimal acceleration. This of course leads to a totally different optimal passing and goal kicking skill.

In order to make future implementations of heterogenous players possible we added an array of player types to store the different player types. Also for each player we could store which player type it actually is. Since our online coach doesn't make any player substitution and no guesses about the opponent player types, which should be done by the coach, because he has data which has no noise added to it by the server, we did not check out how good our adaption of heterogeneous players is. We only checked whether the system works at all with heterogenous player types.

Quite a lot of these values are used in the functions that make predictions of the world's future, for example who is first to the ball. To not have to change the whole access to the world model, we assumed, that every player is of the same player type as our own player which is something of a hack and rather rudimental. Still, this works quite well in most situations, but can under some circumstances lead to misinterpretation of the world state.

To actually use our code with heterogenous players you have to make changes to the world model and to the skills. You have to write a wrapper class, through which you get access to the values of the different players and you have to write a module that makes assumptions about which player is which player type just in case your coach is not telling you and you also have to add this to the communication process.

We had a problem with the timing and the client/server communication. We are not sure whether it was a problem of the UDP-protocol or whether it was a bug in the RoboCup-server, but we did not get messages in the right order. Our problem was that we did not get the `init()` messages as the first response from the server (not all the time). So we had to store the messages and bring them in the right order to avoid missing any initial message. Because UDP does not care whether you actually receive a message, you have to care about your message stack in your client. We used a dequeue for the implementation, which is a container class of the STL of C++.

The code example (`netif.C`) can be found in appendix E

At first we tried to learn the new skills which should have given us a better performance than the CMU skills. The CMU skills are not that bad, but they can be improved. You can find more about our learning approaches in the next chapter. Unfortunately, we ran out of time and didn't have the time to improve the skills by hand. This is certainly a task that remains to be done.

## Chapter 3

# Learning

### 3.1 Motivation

In former competitions of RoboCup there have been several approaches to apply machine learning techniques to the RoboCup domain. Most teams in the competition in 2000 in Melbourne focused on improving the low level skills with help of machine learning techniques. One example is the team Karlsruhe Brainstormers[15] whose low level skills have all been learned using reinforcement learning. In comparison to other teams, this team had significantly better ball handling and interception skills. The team was able to get to the ball about 10 percent faster and they played the ball with a higher accuracy than teams which used the un-tuned hand-coded skills from the freely available CMU source code. This resulted in much less ball losses when playing passes. So we decided to focus on robust low level skills first to build a stable basis for our higher level skills.

Another motivation for the use of machine learning techniques was our interest in the field of new AI, such as artificial life, evolutionary computation, genetic algorithms and neural networks. We had minimal experiences in implementing and experimenting with such systems and we wanted to improve our skills and get the experience needed to successfully apply these techniques.

### 3.2 Problem Structure

As already mentioned, we focused on improving the low level skills. The first two basic skills we needed were `goto-ball` and `kick-ball`. The goal state for the `goto-ball` skill is simply to get into a position that allows controlling the ball. For the `kick-ball` skill the goal state is defined as a situation, in which the ball leaves the kickable area (the area where the ball is controllable for a player) in a given angle at a specified velocity. These two simple tasks are prototypes of most problems that a player has to deal with in RoboCup. The player has to find actions which, from a given situation, lead to a situation which satisfies the constraints given by the goal state definition. Thus the learning algorithm has to map situations to actions.

### 3.3 Potential Learning Methods

Our objective was to find a machine learning approach that is able to map situations to actions with minimal knowledge of the world. Our plan was to specify goal state and situation information together with a set of possible actions. The learning system should then be able to find a solution to the problem with this knowledge only. We provided the system with minimal knowledge in order to avoid influencing the process of finding the optimal solution. This way the system should be able to find solutions we didn't think of before. This constraint forced us to exclude all supervised learning methods. Consulting the literature left us with three different learning methods which were able to solve problems with very little information [7].

- Classifier Systems
- Evolutionary Systems and Genetic Algorithms
- Reinforcement Learning

It is common to all three of these learning methods that they merely need a reinforcement signal. The reinforcement signal is generated by the environment based on how good the task is solved. In our case the definition of good could be, how many steps were needed to get from the starting situation to the goal state. Individuals able to solve the task in less steps receive a more positive reinforcement signal than individuals needing more steps.

Classifier systems operate on a set of rules. These rules consist of two parts, a condition part and an action part. If the condition part is satisfied by a stimulus presented to the system (e.g. the situation description), the action part is executed. The action part may activate an action directly or send another stimulus into the system by which other rules are activated. It is possible for more than one rule to be activated in one time step. By always selecting the first rule that is active it is not guaranteed that this rule is the best rule possible. Here a smarter selection mechanism has to be found. A module is introduced, to assign a fitness value to each rule. The fitness value is estimated over time. Every time a rule is activated, a small amount of the fitness value is "paid" to the module. After a goal state has been reached, every rule involved in solving the task is rewarded relative to how many steps are still needed after execution of the rule. This way rules leading directly to a goal state get a higher amount of fitness than rules executed at the start of an action sequence. By assigning a fitness value to each rule, the selection mechanism becomes very simple: If more than one rule is active, the rule with the highest fitness is executed. A rule set which is able to solve a given task, is generated by introducing new rules using a rule discovery module, interacting with the environment and assigning the fitness values. In a classifier system every rule has to be evaluated once in every time step and the rule set is not limited in any way. Therefore it cannot be guaranteed that the system is able to react in a fixed amount of time. Successful application of the learned skills in a game like RoboCup is dependent on in-time reaction, in our case 100ms. That's why we decided to use another machine learning technique for learning the skills. A method with a fixed response time would be desirable.

As mentioned above, another method to achieve the mapping from situations to actions is Evolutionary Computation and Genetic Programming. This method is a general optimization technique, so it can be used to optimize the weights of a neural net that chooses the appropriate action in a specific situation. In this approach, the neural

net is coded as a vector, called chromosome. Each information coding unit on the chromosome is called a gene. From randomly initialized chromosomes, the first population of neural nets is “grown”. In the next step, the nets are tested on how well they solve a given problem. With this information, the reproduction probability is calculated, in a way that an individual that performs well has a high reproduction probability and others have a lower probability. After that, the offspring is generated. The chromosomes of two individuals are recombined and perhaps some of the genes mutated to form two new individuals whose chromosomes then consist of genes from each of the parent individuals, possibly mutated. Now the new population is tested again, the reproduction probabilities are calculated and so on. After some generations, the individuals’ chromosomes should converge and the produced neural nets should be able to predict the actions correctly.

The last method we considered as an alternative is Temporal Difference Learning. In this Reinforcement Learning approach, a function approximator is trained, in a way that enables it to predict the exact length of the path to the next goal state, given any situation. This is done by training, for example a feed-forward neural network with back-propagation, whose input is the description of the current state, and whose output is one value that represents the steps needed to reach a goal state as a sum of fixed per-step-costs. The net’s weights are initialized randomly. Then the net is trained with the following equation,

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \min_a Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$$

where  $\alpha$  is the learning rate and  $\gamma$  is a discount factor.  $Q(s_t, a_t)$  are the predicted resulting costs from the situation  $s_t$  with the action  $a_t$  to a goal state. Analogically  $\min_a Q(s_{t+1}, a_{t+1})$  are the minimal expected costs from one time step in the future. In other words, the action  $a_{t+1}$  is the best action in the situation  $s_{t+1}$ . If the state  $s_{t+1}$  is a goal state there are no further costs whereas in a failure state the costs are infinite. In this approach the differences in time are used to train the function approximator to predict the remaining costs for reaching the goal. Temporal difference learning has successfully been used to solve complex continuous and discrete problems in various domains. Additionally, we had the chance to ask Martin Riedmiller for experiences and problems in temporal difference learning because he held a seminar about intelligent robot controlling, including reinforcement learning, at our university. Thus we decided to use temporal difference learning to learn the low level skills.

### 3.4 Temporal Difference Learning

In this section we describe the Temporal Difference Learning method in more detail. The simplest approach in learning from temporal differences is Q-Learning developed by Richard S. Sutton [20]. It is a derivate of value iteration, known from the dynamic programming domain. Value iteration is a table based approach. For every possible situation in the world, there is one entry in the table. In the beginning, the table entries are all zero. Then the table is updated every time-step with the following equation:

$$V_{k+1}(s) = \min_a E\{r_{t+1} + \gamma V_k(s_{t+1}) | s_t = s, a_t = a\}$$

for all  $s \in S$ , where  $S$  is the set of all possible situations. The value of the state  $s$  after one iteration ( $k+1$ ) is the cost for a state transition plus the minimal expected cost for the resulting state, with goal states having no future cost and failure states having

infinite cost. The updating process is guaranteed to converge after the updating process has used every action in every state. It becomes obvious, that value iteration is not tractable in a large state space with a huge amount of possible actions because the table grows proportional to the size of state-action pairs.

Let's step back to Q-Learning with a parameterized function approximator. The objective of this approach is to get a compact representation of the value function e.g. a neural network and to utilize the generalization of the function approximator for continuous learning tasks [21]. The first variant we started with is the simplest variant of Q-Learning that required a complete model of the world  $J^*(s, a')$  which is a function that maps a situation and an action onto a situation reached by using action  $a'$  in situation  $s$ . The equation to calculate the Q-value for a state is:

$$Q(s) = r(s, a') + \min_{a'} Q(J^*(s, a'))$$

The Q-value of a situation represents the cost arising on the way to a goal state.  $r(s, a)$  are the cost for each state transition, in our approach, we assigned the same cost to each action. In other domains, it might be possible that some actions produce higher cost than other actions. In our case all actions are treated equally. The training procedure for this first approach in pseudo-code looks like this:

```

create set of training-situations S
while Q(x) didn't converge
{
  choose training situation s randomly
  while (s != goal state)
  {
    for all actions a
    {
      calculate s' = J^*{s, a}
      calculate Q(s')
    }
    select action a' that produces minimal cost
    update Q(s) with Q(s) = r + min_{a'} Q(J^*(s, a'))
    s = J^*(s, a')
  }
}

```

As mentioned above, one of our learning tasks was to reach the moving ball in minimal time. We tried to train a feed-forward neural network to find the correct action in each situation presented to the network as input. Since the coding of the situation for the neural network seriously influences the possibility for the network to find the correct parameters for approximating the value function, we adopted the coding from the Karlsruhe Brainstormers, who released their source code to provide an alternative basis for starters in RoboCup. We recognized that with this simple approach the results were not satisfying. The neural network obviously learned in the first episodes, but then the learning process didn't improve the action selection any more. We found out that the exploration resulting from the weight changes wasn't broad enough for learning this task. After inserting an  $\epsilon$ -greedy action selection the learning performance increased, but the behavior was only suboptimal. In some situations the net selected actions that resulted in a similar situation and not in a better situation. Even more training episodes didn't improve the performance. In fact, this caused the function

approximator to decrease in performance. We found many articles which dealt with the problems of combining a parameterized function approximator with Q-learning. Most authors mentioned that if function approximation in Q-learning is achieved with a neural net instead of a lookup-table, convergence to a global optimum is possible only by chance. Some of the articles provided alternatives to Q-learning whose results should be more reliable.

### 3.5 Grow Support

One of the promising methods was *Grow Support* developed by Justin A. Boyan [3]. In his paper he points out that an iterative process like value iteration in combination with a general function approximator can lead to diverging results. He presents a method that heavily makes use of the generalization of the approximator, instead of forcing the approximator to change the parameters to directly fit the one-step updates estimated by an iterative process. That way the value function should be approximated more reliably.

Boyan's method is based on *rollouts* rather than one step updates. These rollouts are costs of paths, generated by following the greedy policy given by the function approximator. If the greedy policy, from a given starting situation, reaches a goal state, the estimated costs are returned. If the greedy policy doesn't lead to a goal state after a given step-count, infinity is returned. Thus, the correct cost for the state or the misleading of the current policy is returned. If a goal state has been reached, the starting situation together with the cost could be added to a set, called *support-set*, on which the approximator is trained. Given this idea, the rest is straightforward. A set of situations  $X$ , sampled from the continuous state space, is defined and initially, the support-set is empty. Next, the function approximator is trained on the support-set. Then if there are states left in  $X$  and the support-set didn't stop growing, rollouts are performed for every  $x \in X$  a time. If a rollout was successful, the situation-cost pair is added to the support-set. The generalization of the approximator optimally causes many successful rollouts in one training episode by generalizing over a region of the state space. If all  $x \in X$  are processed, the procedure restarts with retraining the function approximator. If the support-set stopped growing, or all previously sampled situations have been added, the approximator has converged and the value function has been approximated. To make this clearer: Here is the main procedure in pseudo-code, given the support- set  $SUPPORT$ , the set of sampled points  $X$  and the approximator  $FIT$ :

```

X := points sampled over the state space
SUPPORT := {}
repeat
{
  train FIT to approximate SUPPORT
  for each state xi in X
  {
    c := argmin(a) [ COST(xi,a) + RolloutCost(NextState(xi, a),
      FIT) ]
    if c is not infinite
      add <xi,c> to the training set SUPPORT
  }
} until SUPPORT stops growing or all point in X were added to

```

the support set

After implementing this method, we trained it on the task of going to the ball, as described above in the context of temporal difference learning. In our case, the training procedure inserted only a few points to the support set, and terminated after a short learning phase. We then tested this approach with larger neural nets, because we thought, the net used in the first run wasn't able to fit the training data, due to its small hidden layer size. Since the training runs with larger nets haven't shown a much better performance, we dropped this approach and implemented another one.

### 3.6 ROUT

The next method of approximating the value function was the enhanced version of grow-support, ROUT [4]. Boyan developed this method on the basis of the grow-support algorithm, to eliminate the negative properties of grow-support. One of these unwanted properties was, that in very large state spaces, the algorithm needed a lot of sample points which, one after another, were inserted in the support-set. So, training in large state-spaces took much time, even if the problem was very simple to approximate e.g with only a few of the points in the support-set. That's where this method focuses. Instead of using the generalization capability of the function approximator to add as many sample points as possible, the generalization is used to find points in that region of the state space, where the generalization starts to fail. Then for these points the correct values were estimated and the training samples were added to the training set. With this technique, only those training samples were added, which are located on the frontier of regions where the approximator predicts correct values and regions, in which the approximator predicts incorrect values. That keeps the training set as compact as possible. And in a successful run, the regions of correct value prediction grow from the goal backwards. Given the starting points  $X$  the training set  $SUPPORT$  and the randomly initialized function approximator  $FIT$ , the learning procedure looks like this

```
SUPPORT = {}
repeat
{
  for each state xi in X do
  {
    s := HuntFrontierState(x, FIT)
    add training sample <s, one_step_backup(s)> to SUPPORT
    retrain FIT to fit SUPPORT
    if s == xi then mark xi as ``done``
  }
} until all start states are marked as ``done``
```

The one-step-backup referred to in the pseudo-code is simply a fixed per step cost plus the expected minimal cost. The procedure HuntFrontierState briefly described above in pseudo-code given the current state  $x$ :

```
for each legal action a do
{
  repeat up to H time
  {
```

```

        generate trajectory T from x to termination, starting with
        action a let y be the last state on T with Bellman
        residual > epsilon if y not empty and y != x, break out of
        loops and restart with HuntFrontierState(y,FIT)
    }
}
// reaching this point, the subtree of x is deemed selfconsistent
// and correct
return x;

```

Even in this enhanced version, the training didn't succeed. After some successful episodes, the greedy policy didn't lead to goal states anymore, with the result that only a small region in the state space was approximated correctly and in other regions of the state space, the approximator predicted misleading values. In the paper, Boyan only mentioned the problem, that if the approximator is not capable of fitting the value function, the support set grows constantly without growing the support region backwards from the goal.

### 3.7 Results

Since we re-implemented our learning method every time we realized that the task wasn't solved sufficiently, we didn't spend enough time in collecting data and analyzing it. Looking backwards, it would have been better to analyze the problems in detail to get a deeper insight what kind of problems occurred in approximating the value function in the different approaches. Then we possibly would have been able to solve these problems with help of literature. The way we worked was more or less a trial and error search for a method that learns the needed skills without any problems. One reason for that was that we didn't expect, that implementing and using that type of learning requires a lot of experience. The less experience there is available, the more time is needed to gain that experience. In our time plan we included one and a half month for implementing and training the skills for our team. That definitely was a phase too short, because after we realized that we won't get a stable learning system or perfectly learned skills within that period, we started to implement other training scenarios and even searched other methods to solve the given tasks, where probably other kinds of problems occurred.

### 3.8 Further Work

Further work in this domain should include the topic mentioned in the previous section. The occurring problems have to be identified by analyzing the data collected during the learning process. An excellent book in which various techniques for analyzing data are described in is [10]. One aim would be to use other function approximators, which have been studied in more detail and used successfully in a broader field. One example would be a linear function approximator that has been used in various robotic tasks, the CMAC. Tsitsiklis [22] mentions in his paper, that nonlinear function approximators in combination with temporal difference learning lead to diverging results whereas linear approximators converge to an optimal solution.

Another interesting domain in reinforcement learning is the application of evolutionary computing and genetic algorithms to sequential reinforcement learning tasks.

There are many new promising approaches in evolving neural controllers that find solutions more reliable than the temporal difference approach.

# Chapter 4

## Playtree

### 4.1 Introduction

Our first approach to realizing the agents' decision making module was to implement a handcoded decision-tree-like structure, the "playtree", in order to have a running team of reactive agents as soon as possible. The stepwise refinement of the basic agents' behavior promised to be a good way towards getting a deeper understanding of what makes a good RoboCup agent.

A decision tree is a set of rules represented in a tree-structure. Nodes represent questions or conditions, querying a particular set of data (the input), while branches leading to nodes on the next level are labelled with answers to those questions. Starting at the root node and answering each node's questions along the way, the traversal of the tree eventually reaches a final node. Final nodes (leaves) are not labelled with a question but with a value or action that is regarded as the system's output.

In the playtree, the input consists of the current state of the world model and internal state variables, while the output is given in the form of action commands, i.e. calls to high level skill functions. In addition, the agent's internal states may be altered at every point in the tree's traversal, e.g. to avoid the repeated evaluation of time consuming queries to the world model or to realize the execution of simple plans in the form of short action-sequences whose execution takes more than one simulation step.

The playtree is implemented in the form of C-functions that correspond to its subtrees and consist of conditionals (`if...else`, `switch`) whose conditions are calls to world model functions and whose actions are calls to other subtree-functions or, on the lowest level, calls to high level skill functions.

Each player has his own copy of the playtree and in every server cycle (simulation step) calls the main playtree-function, which leads to a situation-specific traversal of the playtree and eventually results in the execution of the respective action command. Thus, the players are reactive agents, i.e. most of the time they only choose one action to be executed during the current cycle on the basis of the current state of the world instead of pursuing any explicit goals and planning action-sequences to reach them.

Due to the goal that each player should be able to take over each possible role when it is required, there is only one version of the playtree for all players. All players except the goalkeeper, who has an own goalie-subtree, theoretically behave in the same way when they encounter the same situation. The only aspects in which two given fieldplayer agents differ are their internal states and their environments which trigger

the choice of a particular action.

The idea behind this design was to build the agents' behavior in a bottom-up fashion, starting with rough distinctions like the current playmode or whether the agent is in ball possession, and choosing simple behaviors like getting the ball or kicking it in a certain direction and then refining the behavior step by step by replacing calls to more general behaviors with more detailed queries about the state of the world and the respective, more specialized actions, leading to a tree that is growing, i.e. branching out more and more, with our growing experiences and skills in formalizing the needed knowledge about the domain. In the time needed to identify the crucial situations occurring in a game and to formalize the appropriate conditions, the required high level skills can be developed by a combination of lower level skills or by means of machine learning.

Apart from its extensibility, the playtree has the advantage that its modularity facilitates the independent development of different subtrees. That way, different parts of the behavior can be implemented independently and later be adjusted to work together.

The figure below shows the rough structure of the playtree, the following sections of this chapter take a closer look at it, starting with the part that implements the field-players' behavior and then explaining the part that specifies the goalkeeper's behavior.

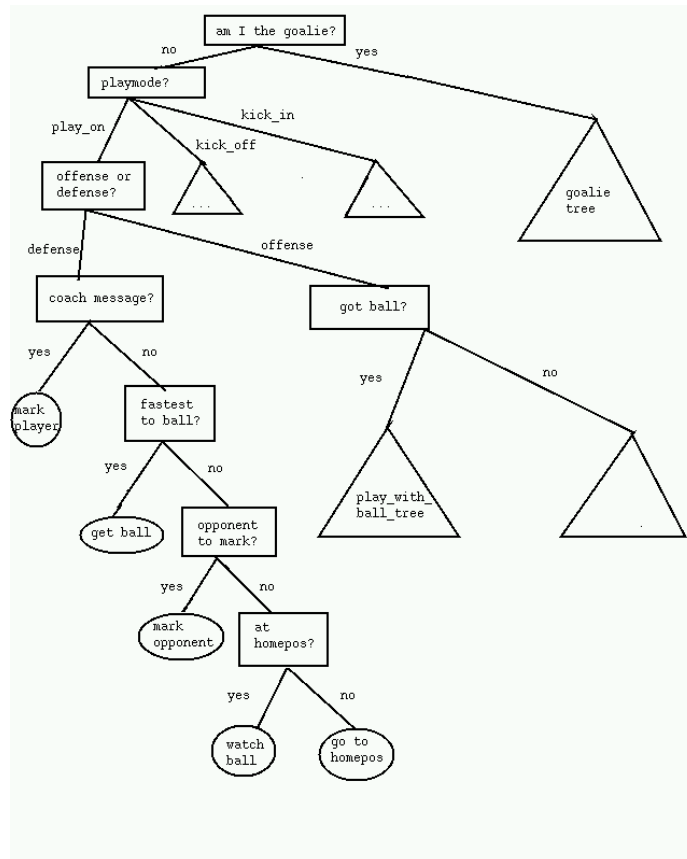


Figure 4.1: A rough sketch of the playtree.

## 4.2 Formations

As the concept of formations, i.e. the mechanism for the strategic positioning of the players on the pitch, is important for the understanding of the detailed description of the inner workings of the Playtree that follows in the next section, it will be briefly explained in this section.

One solution to the problem of the team-wide strategical positioning of the players would be to let each agent determine a strategically optimal position in every action cycle, taking into account the current game situation, the team's strategy and the positions of their teammates, probably using communication to negotiate with them.

This solution has the disadvantage that it is rather complicated and thus error-prone, and that it consumes precious CPU time.

In real soccer, explicitly agreed formations are used for the strategical positioning of the players, telling them where about to position themselves during certain phases of the game, e.g. defensive or offensive play in either their own or the opponents' half. Formations give the team the ability to quickly react as a whole on changes in the game, provided that each player knows his position in the current formation and which formation to switch to in a given situation. The players only have to adjust their current positions according to their role in the current formation instead of reasoning about their strategically best positions all the time.

Another advantage of using formations is the fact that expert soccer knowledge can be easily formalized by specifying a set of formations and the rules when to apply which.

We implemented the concept of formations by providing formation data shared by all the agents, combined with a mechanism for switching the current formation and a behavior that makes intense use of the positioning data found in it. To enable the agents to share the same set of data, it is, in the form of formation records whose structure is described below, externally stored in a configuration file which is read by each agent at the beginning of its lifetime and which has a syntax that facilitates editing formation data.

A formation record simply consists of a formation-identifier that uniquely identifies the formation, and ten positioning records, one for each player or role in the formation.

Apart from a position-identifier, positioning records have three attributes, as illustrated in the figure below:

**HomePosition** is a point on the pitch that the agent regards as its default position and as a starting point for its individual positioning, which is based on the local situation.

**HomeRange** is the radius of a circle around the HomePosition.

**MaxRange** is the radius of a bigger circle around the HomePosition. HomeRange and MaxRange represent horizons for some of the agent's perceptions and actions.

This more centralized way of building formations only works if the mechanism for choosing the current formation is the same for each player and is only dependent on global information, i.e. information that is accessible to each player. To achieve this, the skill sets are extended by `set_current_position(formation_id, position_id)`, an action command which causes the agent to set its current internal positioning data to the data found in the respective positioning record in the respective formation record.



Figure 4.2: Formation data (shown for one player only).

At this stage of development, the `position_id` is simply the agent's uniform number, so that the only thing the agent needs to know is the current `formation_id`. When the global situation of the game changes (e.g. a switch from defensive to offensive play takes place), certain rules in the playtree that only depend on globally accessible information tell the agents to execute the appropriate `set_current_position()`-command. As this information is the same for all the players, this results in a team-wide change of the current formation.

Apart from the advantages mentioned above, team-wide formation knowledge offers more opportunities. Formations give the online coach the power to strategically intervene by simply switching the current formation according to global information about the opponents' positions or strategy.

Or the agents can, for example, adjust their current `HomePositions` according to the ball's position, so that the whole team automatically follows the ball while still building a formation. The resulting `HomePositions` serve as a starting point for the agents' own local decision where to position themselves within their `HomeRange` or `MaxRange`.

### 4.3 Strategy

To simulate a real world of soccer we have tried to take over some strategies for our agents. The main thing is the separation between the play in the offensive and defensive play modes. An offensive play mode means that our team controls the ball and by contrast defense is the situation whenever the opposing team possesses the ball.

For the different strategies in these two situations we also have to try to realize all our knowledge about the real soccer through our agents.

The first part of this section deals with our agent's behavior during the two parts of the offensive play mode. There is a description about the different decisions the agents make. After that the main actions of our offense is explained in detail. The next part illuminates our agent's behavior and its main options during the defensive play mode. Furthermore the details of the main actions of our defense is given. Then there is a survey of the behavior when the team is in the kick-off situations. This chapter ends

with a conclusion about the problems which we had during the development of our team.

### 4.3.1 Offense

This situation is divided into two parts which describe the different behavior of the agent. The first contains the tasks for the agent which doesn't have the ball and the second one shows the play of the ballowner.

#### Agent Without Ball

The behavior of the agent depends on three main decisions. The first option checks if the agent is chosen as the ballreceiver after a pass. In this case he goes to the position where the ball is expected and tries to get it. The information about the estimated position of the ball and the uniform number of the ballreceiver are communicated by the ballpossessor. Therefore the player will know that he will be the passpartner. Sometimes the messages between the players are lost so none of the agents take the responsibility for the pass. There is an alternative where the fastest agent goes to the ball and tries to get it. If the situation of the agent is such that neither of the two preceding decisions fit he moves to a position to avoid the opponents in order to give the possibility for a pass.

#### Get Ball

Our agent uses the procedure `get_ball()` from the CMU-Code to get the ball. The criterion which action is executed is the movement of the ball.

If the ball moves then the agent tries to intercept the ball. A function computes a point on the way where the ball will be and the agent will go to that position. Otherwise if the ball doesn't move the agent will go to the position where the ball lies.

#### Free Run

Which position the agent will move to in order to avoid the opponents is determined by an evaluation of different positions. A function `getFreeRunPos(...)` checks the number of the opponents which are on the way from the computed positions to the position of the ball. The agent goes to the position with fewest opponents because this will be the best position where the ballpossessor can pass the ball to.

The evaluation considers eight possible positions where the agent can go to. The position is a point which is

- on the end of the circle of the homerange of the agent.
- between the homeposition of the agent and the position of the ball.
- between the homeposition of the agent and the goal of the opponent.
- the homeposition of the agent.
- the current position of the agent.
- one meter from the current position of the agent.
- between the current position of the agent and the ballposition.

- between the current position of the agent and the goal of the opponent.

### Agent With Ball

If the agent is the ballowner there are four main options. At first the agent looks if there is a possibility to score a goal. If there is no chance for a goal then the agent chooses a teammate for a pass. In case he finds a suitable passpartner the agent communicates the estimated position of the ball and the uniform number of the player who will get the ball. Then he kicks to that position. If there isn't a good passpartner the agent tries to dribble the ball in a direction where there are not too many opponents. Sometimes it is dangerous to dribble the ball because it is too easy for the opponents to get the ball. In that case the agent kicks in that direction. This will give the teammates a chance to get the ball or at least the ball will go closer to the goal of the opponent.

### Goal Kick

There is a procedure `goal_kick()` that checks the possibility to score a goal. The agent only kicks the ball towards the goal of the opponent if the distance of the agent to the goal is equal or smaller than 19,5 meters.

The goal line of the opponent is divided into 25 points with the same distance between the points. Lines between the position of the agent and these points are drawn and are checked if there are opponents on them. The agent kicks the ball towards the goal if there is a line with no opponents on it. The lines are checked from the outside to the inside so there is a chance of a good goalkick into the corner of the goal because the chance of the goalkeeper to catch the ball is more improbable in that case.

### Passing

In order to get a convenient teammate for a pass there are two strategies. The first strategy of the behavior our agents is realized in the decisiontree by the function `passpartner()` and the second one describes the behavior our SFLS- team uses in the function `bestPasspartner(...)`.

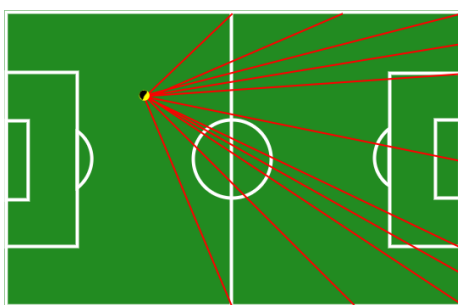


Figure 4.3: Decisiontree Strategy: The ballowner is on his own half.

The two strategies are based on the same fundamental idea. In order to find the direction the ball will be shot into the ballowner orientates himself at points in the field or at the fieldline. Lines between the position of the ballpossessor and the different points are drawn and after that are checked for the number of opponents that are in a

cone along the lines. The lines with the fewest opponents are checked for the number of teammates which are around the lines. If there is at least one teammate then the position of the teammate (or teammates) is projected on the line.

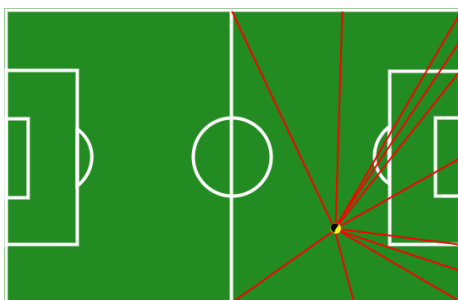


Figure 4.4: Decisiontree Strategy: The ballowner is on the opposing half.

There is a function `positionPassValue(..)` which evaluates the different points with three criteria. The first one is to look at how many opponents are around the point in a radius of 2 meters. The next value is computed for the position of the point compared to the length of the field. That means the idea is to play the ball outwards in the back of the field and inwards from the middleline on. The last criterion is the distance to the opposing goal. These three values determine the choice of a point on a given line and thus the teammate. After this is done there is another check of the opponents but only up to the projected point of the teammate. The direction with no opponents is chosen and the ball is played. The teammate whose point is projected on the line is the suitable partner for the pass.

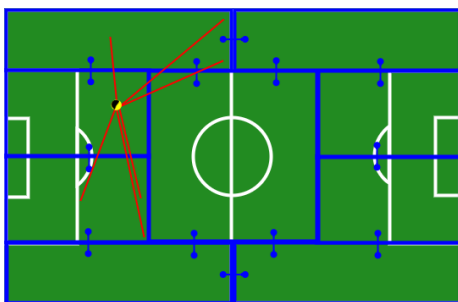


Figure 4.5: SFLS Strategy: The ballowner is on his own half.

Now, the difference between the two strategies are at first the positions of the base points. In the decisiontree strategy fifteen points are placed on the fieldlines at the opposing side which should help to find the direction (fig. 4.3, Fig. 4.4). By comparison in the SFLS strategy the pitch is divided into regions in which the points are fixed (Fig. 4.5). The regions are the second difference. Into which region the ballowner will play the ball is determined by his position because not all regions are joined (Fig. 4.5).

### **Dribbling**

To dribble the agent uses a procedure `kick_ball(...)` from the CMU- Code. The agent shoots the ball a little in front of himself and intercepts it again immediately. The direction where the agent is dribbling to is chosen by the same strategy as the pass but the line might not be free of opponents. If there is an opponent who wants to get the ball the agent tries to rotate the ball around himself so the opponent can't get it.

### **Clearance Kick**

A clearance kick is done if there are opponents around the agent in a radius of 2 meters. Then it is more dangerous to dribble the ball. The agent kicks the ball along the best line. That means the line along which he would dribble normally. The agent uses the same procedure `kick_ball(...)` but the shot is much harder.

## **4.3.2 Defense**

There are four main decisions which determine the action to be executed by the agent.

The first thing that the agent will do is look for a message from the online coach. It is possible for the online coach to communicate an opponent who the agent should cover. If there is no such message then the fastest agent goes to the ball and tries to get it. If the agent isn't the fastest player then he looks for an opponent to mark.

In a situation where there isn't an opponent to cover for the agent (for example all opponents are covered) then he goes to his homeposition and chooses the best point in his homerange to observe the ball.

### **LookForBallFromHomePos**

If the agent is not responsible for an action as to get the ball or to mark an opponent then he moves to a point which lies between his homeposition and the position of the ball. The point is situated on a circle around the homeposition so the distance to the homeposition is equal to the radius of the homerange.

### **Cover**

The function `getOpponentToCover()` is used by the agent to choose his opponent to cover. An opponent which can be covered should be in the maxrange of the agent who will cover. So at first there is a check if there are any opponents in his maxrange.

Those opponents are rated according by their current position. The values which determine the opponent are computed by the function `opponentWeight(...)`. There are three distances which are considered. The first one is the distance from the position of the opponent to the homeposition of the corresponding agent. This value is the smallest because it is the least important. The next value is the distance of the opponent to the ball. Further the most important value is the distance of the opponent to our goal.

The next step is to control the opponents whether they are already covered by another teammate. The check begins with the best opponent, that is the opponent with the highest value. If none of the opponents are in the maxrange of the agent then he looks at the closest opponent to our goal and whether he should be covered. If that isn't the case then he looks at the opponent closest to himself and checks if a teammate is near this opponent or not.

There are three different kinds of marking. The first one is to cover our goal. This situation is realized if the distance between the opponent to cover and the goal is smaller than thirty meters. The agent then goes to a point between the opponent and the goal. Ideally, there is no possibility to shoot at the goal. The second case is to place the agent between the opponent and the ball, so, the agent prevents a pass from the opposing ballowner. If the agent doesn't know where the ball is then he goes directly to the opponent.

### 4.3.3 Other PlayModes

The behavior of the agent which was described so far shows the play during the `play_on`-mode. Of course, there are other situations during a game of soccer which the agent has to master.

#### **Kickoff**

The kickoff is done by the player which is the closest to the ball and therefore the closest to the kick-off point. Mostly the agent with the uniform number eleven goes to the ball and passes the ball to the teammate with the uniform number five. It turned out to be that the position of this player is the best to begin the game. To kick the ball the agent uses the procedure `pass_ball()` from the CMU-Code.

#### **Free kick, corner kick, kick in, offside kick**

For these situations the agent uses the same procedure `my_kick_tree()`. One agent goes exactly to the ballposition. To find a teammate to receive the ball after the kick the procedure `passpartner()` is used. In case there isn't a suitable `passpartner` a second agent has already gone into the direction of the ball and stopped 5 meters short of it. So, the agent who wants to play the ball has a teammate to safely shoot the ball to. If the agent is neither the closest nor the second closest to the ball then he tries to avoid the opponents. For this behavior of the agent the procedure `getFreeRunPos(...)` is employed.

### 4.3.4 Conclusion

To realize our simulated soccerteam we had to overcome some problems. Transferring the theoretical ideas into practice turned out more difficult than we thought.

At first we underrated the noise which the server introduces into the game, e.g. the deviation of the positions of the player or the position of the ball. Often we had trouble with the position of the teammates or the opponents because they stood on a different position than we assumed. Obviously, for instance to kick the ball to a teammate exactly was a difficult action. Furthermore the possibility to compute the positions of the opponents exactly didn't exist, therefore they were again and again in the way during a pass action.

Another difficulty was handling the information about the visibility because the agent doesn't see the player or the ball if they are too far away from him or not in the viewcone of the agent. So, the exactness of the knowledge about other players and the ball decreased gradually. In those situations the agent had to estimate the positions. Of course they aren't exact that way.

These problems are introduced by the server because the representation of the game should be realistic. To filter out the server-introduced noise in the information about the game required much experience.

To find a good strategy, e.g. for passing the ball or for covering an opponent was another difficulty. We had to test different strategies. To find a suitable passpartner, e.g. we have first tried searching on the whole field. That required a lot of computation time because the ballowner had to consider all teammates. Another problem was that the agent passed back at the wrong moment or a group of agents started passing back and forth to each other in front of our goal. So, the ball seldom got into the opposing half and we had to deal with many risky situations.

Furthermore there were problems with all agents going to the same region, e.g to cover the same opponent or to get the ball. This difficulty was solved by the implementation of conditions, e.g. not going to the opponent if there already were enough teammates around a given opponent. At first we didn't consider those small but very important aspects.

## 4.4 Goalkeeping

This section will describe the agent of our team responsible for goalkeeping. It will explain why we chose to divide the decision-making part of the code (playtree) and why we created a playtree of its own for the goalie. The remainder of this section is organized as follows: There will be a short introduction on the tasks of the goalie and on why we made the decision to treat this agent differently. The next part will deal with the knowledge of the ball position and with watching the ball constantly. Then we will describe our positioning and movement concepts for the goalie, paying special attention to the importance of these two for good goalkeeping. The last two parts will treat catching and handling the ball, and there will be a short evaluation of our goalie's behavior.

### 4.4.1 Introduction

The agent who is supposed to keep the goal has to be treated somewhat different from the other agents. As in real life, a goalkeeper in general has tasks that differ from those of field players. The obvious ability given to this agent is of course to execute a catch. That, and this agent's more specialized handling of game situations, made us aware of the need to create a separate goalie playtree. This decision evolved in the early stages of the playtree programming. Subsequently, the two playtrees still shared some of the features, but were developed differently in critical areas. These areas include watching the ball, positioning the agent and ball handling after a catch. We'll deal with each of these in the following sections.

Our playtree during `play_on` situations follows these general decisions:

- Check the view width
- Try to catch the ball (or kick it away)
- Try to intercept the ball if it's in our own penalty area
- Try to intercept the ball if it's a shot
- Find a good position

- Scan the field

Of those game situations where the ball is resting, the only two really important to the goalie are the ones after he has caught the ball or when a shot went wide of the goal. All other ones are equivalent to the general playtree of the other agents. A feature that was included in the code but not used was the possibility to evaluate the coach messages.

#### **4.4.2 Watching the ball**

Even more so than other agents the goalie needs to know exactly where the ball is. In critical situations, e.g. when the ball is in the own penalty area or close to it, the goalie agent should watch the ball constantly. Therefore, the first option considered in the playtree is whether to switch the view focus or not. Narrowing the view focus is only advisable when the ball is very close to the agent.

We created two routines that test whether the ball is moving towards the goal. The first, `BallHeadingTowardGoal`, tests for general ball movement and direction thereof, whereas the second, `IsShot`, evaluates all situations in which the ball is moving in our direction. `IsShot` then takes into account the ball velocity and the proximity of the ball to our goal. It turned out to be important to recognize shots quite early, as to have enough time to reach the interception point. But equally important was to avoid false positives, because they led to errant goalie movements, which in turn led to losing sight of the ball and bad positioning.

#### **4.4.3 Positioning and Movement**

##### **Importance of good positioning**

In the RoboCup simulation league, it is especially important to pay attention to positioning. As in real life, it decreases the need for movement (and with it loss of stamina) and increases the chance of intercepting the ball. Assessing game situations, and reacting appropriately to ball position, ball speed, and player positions is an important area of agent development. This is especially true with respect to goalkeeping. If another agent misjudges the velocity of the ball and fails to intercept it, then in the worst case the opposing team will get the ball. If the goalie fails to intercept, the result is usually a goal.

It is important to be in a good position to catch the ball in any game situation, because the ball speed can be so much greater than the player speed. In the simulation league, you also have to take into account that the goal is twice the size as in real life. This means the goalie might have to cover twice the distance. If the goalie fails to be in a good position before a shot, there will never be a chance for him to catch the ball in time. One cannot stress this factor enough — even though it might seem to be a trivial real life observation.

##### **Movement concepts**

The other important factor in goalie design is his knowledge of the position of the ball, as described earlier. To ensure that the goalie always knows where the ball is, it seems important not to move too much (because that might involve turning the neck and so forth). This was achieved by integrating a movement threshold, which made the agent ignore minimal position changes.

Then, if the agent has to move, he should do so without losing sight of the ball. That is, we let the agent move backward if getting to the new position involved a turn greater than a certain threshold. Of course, long distances should not be covered in this fashion, for the agent cannot dash backwards as fast as forward.

### **General positioning concepts**

Obviously it is best to position the goalie on a point between the ball and the goal line. To avoid moving too far to one side of the goal, though, it is best to make the movement towards the sides of the goal more difficult the further out the agent gets. In 'FindGoaliePosition' we use a line parallel to the baseline and 5 meters further into the field to position our goalie. An intersection between the projected ball-line with this 5-Meter-Line is computed. An exception to this rule is the case when the ball crosses the 50-Meter-Line. Then we compute an intersection between a halfcircle around our goal and the projected ball-line. Finding a point does not mean that the goalie will actually move to that point. As mentioned before, marginal position changes are ignored.

### **Situational positioning**

During gameplay this general positioning concept is the default case for goalie behavior. If a situation arises in which the goalie deems himself to be the closest and / or fastest player to the ball, he of course leaves his position to intercept the ball. We also experimented with concepts for positioning of the goalie in case he should come out of the goal to intercept an opponent or to cover as much of the goal as possible. For the former, we used the CMU 'ShouldIComeOutToOpponent' function.

#### **4.4.4 Catching and handling the ball**

For catching the ball we used the CMU function 'goalie\_catch'. We experimented with a routine that tried to delay catching the ball (when it was catchable) for one more cycle and using the gained cycle for movement as to improve catch probability. Even though the server-side catch probability is set to 100%, the goalie can miss the catch because his knowledge of ball speed and ball position is inaccurate. Thus we tried to make up for these marginal errors by trying to get even closer to the ball to be absolutely sure. Unfortunately, we were not able to prove or disprove the validity of this approach.

Sometimes, when a catch fails, it is still possible to kick the ball during the next cycle. We used this extensively to 'get rid' of the ball.

After a catch, the goalie has a few possible options. Because the agent can use the 'move' server command, he can be placed anywhere inside the own penalty area. If there were too many opposing players in the goalie's vicinity, or the agent wasn't able to find a passpartner, he moved again to the other side of the penalty area. The goalie also waited some time (usually 25 cycles) to give his teammates a chance to reposition as good passpartners and to regain stamina.

A similar situation that basically uses the same code is the goalie kick after a shot went wide of the goal, except for the fact that the goalie can't move.

We often ran into the situation that our goalie moved to a spot quite far out in the penalty area, kicked the ball and then the pass was intercepted. This usually led to a goal against us because the goalie didn't have enough time to get back to guard the goal.

#### **4.4.5 Evaluation of goalie behavior**

Watching the goalie was sometimes very frustrating. Often, he misjudged ball speed and heading, and failed to catch the ball properly. This could have been due to insufficient information. Without a reliable defense, the goalie frequently had to handle very dangerous situations in which he didn't really have a chance. We didn't use a routine to commit the goalie to a certain course of action, but rather let him decide each cycle on his action for that cycle. This often led to seemingly confused behavior, as the agent decided on one option, and on a different one the next cycle. In the beginning stages of our programming, the goalie often adjusted his position with respect to the ball, and in doing so turned and lost sight of the ball. This was especially harmful when opponent teams used crosses in our penalty area.

#### **4.4.6 Conclusion**

This section described our implementation of a playtree for the agent who is the goal-keeper. The final version is the result of much experimenting and programming, in the course of which we changed the code quite often. We came to realize that the goalie and his defense maybe need to interact more to improve the handling of potentially dangerous situations. We also somewhat neglected the importance of keeping the ball in sight. We think that most of the goalie's errors were due to insufficient or inaccurate information.

### **4.5 Conclusion**

Implementing a team using the described playtree is straightforward. Partitioning the playtree into modules for offensive, defensive and goalie behavior proved to be useful, because this way people were able to implement with less conflicts. Yet, there are several shortcomings. Integrating coach advice into the playtree is difficult. Also, even small modifications have to be compiled which is time-consuming. Also changes in the code are prone to result in errors. Due to the architecture and error-handling of the CMU-code many last-time-improvements turned out to cause the agents to crash because of missing checks. In order to understand the overall behavior of the agents, one has to skim through several files and many lines of code.

These shortcomings are overcome in our SFL-approach which is described in chapter 8.

# Chapter 5

## Communication

### 5.1 Introduction

In the RoboCup domain agents are able to communicate among each other. They do this via the say-Command. There are relatively strict limitations on what can be communicated. Basically the message an agent can send to other agents is a string of limited length (about 512 characters). This string is also limited as to the characters that are allowed. Only alphanumerical characters and ten special characters may be used. There is also a maximum hearing range that defines how far such an utterance can be heard. Every agent can only issue one say-Command per cycle. An even stronger limitation is due to the fact that each agent can only hear one message per cycle. This implicates that an agent can only hear one of his teammates every other cycle.

### 5.2 Sharing knowledge about the current state of the world

Communication is possible in RoboCup which leaves the question what to use it for. One important aspect of the RoboCup domain is incomplete and inaccurate knowledge of the world, which is due to the sensory limits of the agents. Therefore it makes sense to use communication to somehow overcome or lessen this problem. Since all agents have a different view of and on the world they all have different information in differing qualities.

#### 5.2.1 The protocol and the compression

Inspired by the strong limitations on message size and alphabet we chose to implement a compression mechanism that allows for a maximum amount of information to be communicated. A little reflection on the types of information to be passed on to other agents revealed that about everything could be expressed using numbers. There are integers, floating points, and boolean values. Our compression mechanism lets us define the kind of numerical value, its range, and how many bits its precision should be. This way we can put every available bit to use (see Table 5.1).

name	type	range	precision
defense	boolean	0..1	1bit
Unum	integer	0..11	4bit
xCoordinate	float	-60..+60	10bit
yCoordinate	float	-35..+35	9bit
confidence	float	0..1	9bit

Table 5.1: Examples with type of numerical value, range and precision.

Each agent has its own message object. The message object can be fed a string, decode it and update its values accordingly. And it can be asked to encode all of its values into a string using our binary compression algorithm. The encoding and decoding is done in such a way that the message object that receives a message string is then filled with the very same values as the sending one.

### 5.2.2 The architecture and structure

We designed the message strictly object-oriented to consist of other more specific messages. This way the toplevel message would consist of a header message, two team messages, a ball message, and a strategy message. Each of these messages is again made up of more specific messages. This next example shows where in the message structure to find the stamina value and its confidence.

**Message** (own team, opposing team, ball, strategy)

- header (time, sender)
- **own team (11 mates)**
  - mate
  - mate
  - **mate**
    - \* position (x, y, confidence)
    - \* velocity (x, y, confidence)
    - \* **stamina (stamina, confidence)**
      - stamina (float 0..3500, 5bit)
      - confidence (float 0..1, 9bit)
    - \* neck angle (angle, confidence)
  - mate
  - mate
  - mate
  - mate
  - mate
  - mate
  - mate
  - mate
- opposing team (11 opponents)
- ball (position, velocity)
- strategy (formation, offense/defense, passmessage)

### 5.2.3 Updating the world model from the message

To update the world model from a received message is not a trivial task. There are two trivial ways to deal with an incoming message. You can either completely ignore it or believe everything. While completely ignoring it would render all communication useless, believing everything you hear is seldomly a good idea. This would mean you ignore everything you already know. We therefore have to think of a mechanism that decides which information to keep and which information to update based on the data of the message.

In our world model all data that is subject to change due to the dynamic environment has a confidence-value. This value has a range of zero to one. Whenever anything is observed directly from the environment the according confidence-value is set to 1. In every timestep all confidence-values are decreased. Thus older information has a lower confidence-value.

These confidence-values help a great deal with integrating data from the message into the world model. All the data in the message is communicated with the according confidence-values. To decide which data to keep we can start out with comparing the confidence-values. If both confidence-values are the same we can take into account who told us. The header of each message includes the sender and the time the message has been sent. With this information we can find out whether the sender is closer to the object in question and therefore is likely to have less noisy view.

## 5.3 Communicating a plan

Communication is not all about facts and raw data. It can also be used to coordinate future actions with others. We did this and are going to explain how in the following section. As an example we are going to look at passing.

### 5.3.1 How to express a plan in a message

Passing requires two players and the ball. One player has to kick the ball in a way that his teammate receives the ball at a future moment at a certain position. To inform the teammate what the plan is, the player passing the ball has to tell who he wants to pass the ball to and where he is going to kick it. The rest of the plan is of course implicit. The teammate has to know that he better get to that position and get the ball. The variables in this plan are only the uniform number of the receiving teammate and the position the ball is passed to.

Putting these two in a message is no problem since we already have messages for numbers and positions. For every message a player receives he checks whether his uniform number is in it.

### 5.3.2 Information loss and relaying

Of course in RoboCup agents don't hear every message. There is a maximum hearing range which determines over what distance the agents can communicate. And maybe even more important only the first message that reaches the server is broadcasted (see 5.1). In world model communication that is no big problem since you can establish a protocol that tells each agent in which cycle it can broadcast again. This way you avoid collisions and can make sure that everyone is heard at some point. In communicating

plans like in the passing example things can easily turn out to be too time critical to be handled this way. The worst case being an agent having to wait for over two seconds until it is its turn to talk again. The agent can of course talk no matter if it is its turn or not. The problem arising is that there is no way of telling whether the message will get through or not. To assure this the message has to come in first. This can be easily achieved by introducing a priority for messages. If a message has no priority it waits for its cycle and the normal communication interrupt. If a message has a priority it is communicated at once. This cycle and early in this cycle. By doing this the message has incredibly higher chances of being heard. To make sure the player that is supposed to get the prioritized message actually gets it we introduced a form of relaying. Any agent receiving a prioritized message itself sends prioritized messages for some time. Including of course the crucial information of the prioritized message. These two simple measures (prioritizing and relaying) increased the speed of messages spreading across the playing field tremendously. Without them it often took up to 12 cycles for the message to reach its recipient while it only took 2-3 cycles with these measures.

## 5.4 Promises of a communication using SFL

This binary compression that we are using to get as much information across as possible is of course not the only way to do communication in RoboCup. Another method of communicating would be to send well-structured rules. During the final phase of the project, when we introduced SFL, the thought came up to just communicate SFL-rules. By doing this the agents could clearly tell each other what to do in the language that tells themselves what to do. They could exchange more elaborate information on the world. Of course, this would make it impossible to transfer as much raw data in the same string.

This is of course completely different from the way we used communication but it seems like a train of thought worth to follow. The rules could be integrated into the rulebase as soon as they are heard and thus make the agent act in a desired manner. Problems that are sure to arise are:

- how should a single agent come up with a rule that his teammate cannot but should know about
- does a team that exchanges rules have some sort of hierarchy that determines who should listen to whom
- how should an agent treat a heard rule compared to one that has always been in his rulebase

These problems should not deter anyone but rather show that this is an interesting way to go. And of course, there will be a couple of important questions that are not in the above list.

## Chapter 6

# Logfile Analyzer

### 6.1 Purpose of the Logfile-Analyzer

The logfile analyzer's purpose is to gather information from logfiles of past games. We designed the analyzer in hope to find patterns across a number of games. Our team could then be designed with these findings in mind (to mimic or counteract them). Starting with our task we had approximately 150 logfiles from the last World Championship and the last European Championship. Every logfile is about two megabytes in size.

### 6.2 The Logfile Analyzer's Basis: TimeSlice

#### 6.2.1 The purpose of TimeSlice

Because RoboCup is discrete, it has a finite number (approx. 6000) of frames describing each game. We designed a class which represents such a state. An instance of this class knows all important features of one slice of time in one game. Hence the class-name TimeSlice. TimeSlice has two different approaches to save the data of a slice: an absolute and a regional approach. In the absolute approach the coordinates of all 23 moveable objects (two times eleven players, plus the ball) are saved. In the regional approach the field is divided into regions (the number and the size of the regions is configurable). Within this approach only the number of the region the object is in, is saved. This results in discrete values (over a small value-space) for the objects.

A game is represented as a series of linked TimeSlices. This makes it possible to compute data spanning more than one time frame. There are methods to „look into the future“. This means that a TimeSlice object knows in how many frames a certain action will occur. The actions for which TimeSlice does the calculations are goalkick, pass and ball-loss.

These two domains (positional data and „future“ data) are present to be able to infer tactical information. Combining these domains the idea was to get rules like „if player A is at position B and player C is at position D there will be (with a chance of X%) a goal in Y cycles“. To get such rules we tried to use well-known algorithms. These algorithms are discussed below. But we first take a look at how to use our TimeSlice implementation.

## 6.2.2 Using TimeSlice

TimeSlice is implemented as a C++-class. The main constructor gets all the data to fill the underlying data-structures. The information about the moving objects are passed within the structures `playerinfo_t` and `ballinfo_t` which are defined in `TimeSlice.h`. There are many `get-Methods`<sup>1</sup> defined to access information about the object (like `ballInLeftHalf`, `getPossesingTeam`, etc.). These functions cover the positional data of the time frame represented by the queried object.

To compute the „future” information every TimeSlice needs information about its position in the chain of TimeSlices. This is done by giving the TimeSlice constructor a pointer to its preceding TimeSlice object. Functions to access this „future” information are: `getTimeTillPass`, `getTimeTillGoalkick` and `getTimeTillLosingBall`. They return the number of cycles it will take till the associated action will take place. The computation process takes place in the constructor of the TimeSlice class. Of course when you construct a TimeSlice object you cannot know about the future. So the „future” information is propagated back when it is encountered. This means that if a TimeSlice object, which covers a time frame in which a goalkick happened, is created, the information about the goalkick will be send back to all slices preceding this one. So, in practice, one should only inquire about „future” information if the whole chain of TimeSlices (the representation of an entire logfile) is constructed.

Then there are functions which serialize the data, so it can be written to disk. The usage of these functions depends on what you want to do with the output. E.g. the function `saveC5RegionalPassData` saves the data in a format which can be read by C5.0.

## 6.2.3 A tool which uses TimeSlice: readLog

The main purpose of `readLog` is to read a logfile from a RoboCup-game and create a chain of TimeSlices of it. Then (depending on the parameter given to `readLog`) it does some number crunching and saves the result. You can get usage info by starting `readLog` without any parameters. Here’s an example: if you want to generate data readable by FOIL from the file `test.log` you would call:

```
./readLog -f test.log -x foilpass -X ../../FOIL6/foil6
```

where `../../FOIL6/foil6` is the path to your FOIL executable. This would automatically read the logfile, construct the chain of TimeSlices, compute the values for FOIL (via `foilPassDribbleShoot`) and send the values to FOIL (via an anonymous pipe).

## 6.3 SOM

The first idea we had, was to use a SOM<sup>2</sup>. We used the implementation `SOM_PAK` written by the SOM Programming Team of the Helsinki University of Technology. We used the latest available version which was 3.1. We wanted to use SOMs to cluster the information we had in the logfiles. We hoped to get quantitative data about soccer concepts (like duels, massive dribbling, massive passing, etc.). If we had such information we could build our team with that in mind. E.g. if we would have found that many

---

<sup>1</sup>methods with a void argument-list

<sup>2</sup>Self-Organizing-Map

teams stop passing once they came within 20 meters of the opponents goal, we could have build our defense in a way to counteract this.

The general problem with SOMs is that you cannot calculate the optimal configuration of the net for the domain it is to be used for. You have to run it several times in different configurations and see which one nets the best results.

We also had the problem of choosing the right features to represent a time frame. For the first run we chose the positioning and future information to be used for clustering the frames.

We started with a net consisting of 90.000 neurons (300x300) and a random initialization. We than fed this net with the logfiles from the World Championships 2000. The problem was that it took three weeks (on a Sun Enterprise 4500 with two gigabytes of RAM) to train the net. This meant that our time schedule did not allow us to run tests with a multitude of configurations as was initially planned. We only tried one other configuration: a net with 22500 neurons (150x150). This did not get the desired effect and the time which was scheduled for this part of the project ran out. We now think that maybe we chose the wrong features for representing a time frame. But it is very hard (if not impossible) to chose a representation a priori (i.e. without any tests). So this is another point where we would have gotten better results if we would have had more time to try different setups.

Conclusion: We did not get what we expected from SOM. But this was mainly due to the enormous amount of computing power needed to run the training of the net. Albeit these difficulties we got one important strategically information. This means for us that SOMs might be a good idea if you want to solve similar problems (getting information from data without knowing what information you want exactly) and have the appropriate amount of time and/or computing power.

## **6.4 FOIL**

### **6.4.1 What it is**

FOIL is a tool which gets definitions of datatypes (discrete or continuous) and instances of relations consisting of these types as input. There are two types of input relations: positive and negative. These relations can be seen as examples from an object-space (the negative relations are of course example which are *not* in the object space). FOIL now tries to find one or more horn-clauses which describes an object-space which includes all the positive examples and non of the negatives ones. If it fails to find an exact clause it will try to approximate (minimizing the number of wrong categorized examples).

### **6.4.2 Why we chose it**

We thought that by using our TimeSlices as examples, that FOIL would be able to generate rules covering these examples. We thought that rules could be generated which stated mechanisms or tactics which hadn't occurred to us previously.

### **6.4.3 What we did with it**

We tried to learn three predicates with FOIL: pass, dribble and shoot. We would learn each predicate with its own FOIL run. I will describe in detail how we got the input for

FOIL to learn the pass predicate. The other two were taken similar care off.

The generation of examples for FOIL was implemented in the `TimeSlice` class via the function `foilPassDribbleShoot`. For FOIL, an example is a vector with ten elements: the position of the ball owner ( $x$  and  $y$ ), the number of teammates in a cone, the number of opponents in that cone, the position ( $x$  and  $y$ ) of the cones starting point<sup>3</sup>, the position of the nearest teammate (angle and distance to the ball owner) and the position of the nearest opponent (angle and distance to the ball owner). The starting point of the cone is the position of the pass receiver. The reasoning behind choosing these information was, that the relationship of opponents to teammates in region around a happening pass are vital to the success or failure of the pass (failure means loss of ball ownership).

To analyze a game we would go through the logfile (via the `readLog` tool) and generate a positive FOIL example for every `TimeSlice` object with a `timeTillPass` equal to zero. To get negative examples we assumed a closed world assumption, meaning that there are only three interesting events: pass, dribble and shoot. A candidate for a negative pass example would a positive dribble or shoot example and vice versa.

Before passing the examples to FOIL they were divided into actual examples and test cases. If you supply test cases FOIL is able to give you a plausibility percentage for the clause it generated<sup>4</sup>.

The problems we had with this approach were due to our datatypes. For the positions we needed to floats. Floats are possible in FOIL: they go under the datatype continuous, but this datatype is very sparsely documented. We thought that FOIL would be able to calculate with continuous data. We envisioned predicates like

```
pass (A, B, MatesInCone, OpponentsInCone, E, F, G, H, I, J) :-
...
MatesInCone > OpponentsInCone
...

pass (BallOwnerX, BallOwnerY, C, D, E, F, G, H,
      NearestOppnentX, NearestOpponentY) :-
...
BallOwnerX - NearestOpponentX > 2,
BallOwnerY - NearestOpponentY > 2
...
```

But that was not what FOIL generated. We got very complex predicates with a very low probability tag (most times around 50%, which is just as good as a guess!). There were many floating point constants in these clauses which made them rather useless: a rule which states that a pass succeeds if the ballowners  $x$ -coordinate is 14.56195 was not something we wanted to hand to our strategy group<sup>5</sup>.

We then tried to get away from the continuous values by discretizing them. We created four discreet datatypes in FOIL. A datatype for coordinates ranging from -52 to 52 with a stepsize of two. A datatype for player numbers ranging from zero to eleven. A datatype for angles ranging from 0 to 360 in steps of five. And a datatype for distances ranging from zero to twenty with a stepsize of two. Every atom had to be unique over all datatypes, otherwise FOIL would compare apples to oranges (read: coordinates to

<sup>3</sup>the cone length is configurable, but stayed the same over all examples for a FOIL run

<sup>4</sup>The percentage of the test case space which is covered by the clause

<sup>5</sup>although such a strategy might be easy to implement

player numbers). We did this by prefixing each atom with a letter corresponding to the datatype (a for angles, etc.). Because of that we didn't have numerical atoms anymore and FOIL couldn't use its building comparison relations (<, >, etc.). For this reason we specified a „greater“-relation for each datatype, by simply stating all facts covered by this relation. E.g. the definition of `greaterCoord` for FOIL looked like:

```
*greaterCoord(Coord,Coord) ##
c52,c50
c52,c48
...
c50,c48
c50,c46
...
c-50,c-52
```

The results we got with this approach weren't much better. Using the above method on the logfile of the finals of the Worldchampionship 2000<sup>6</sup> we got the following clause (with a probability of 58%):<sup>7</sup>

```
pass(A,B,C,D,E,F,G,H,I,J) :- A<>E.
pass(A,B,C,D,E,F,d0,H,I,J).
```

An error rate of 42% is very bad. It means that the clause is only slightly better than guessing (which would have an error rate of 50%). And the clause itself doesn't say that much. It means that a pass should occur when the x coordinate of the ballowner is different to the x coordinate of the pass receiver (which is the starting point of the cone), or when the distance to the nearest teammate is zero.

---

<sup>6</sup>FC Portugal vs. Brainstormers 2K

<sup>7</sup><> means „not equal“

## **Chapter 7**

# **Online Coach**

### **7.1 Introduction**

Just like in human soccer it is useful to have someone observe and analyze the game from the outside. Someone who is not supposed to act as fast and as much in realtime as the players on the field and who can provide advice. In the RoboCup simulation league a privileged agent can connect to the server in order to work as a so-called online coach. The ORCA project implemented such an agent which is described in this chapter. The remainder of this chapter is organized as follows. Section 2 gives an overview about the online coach in the RoboCup domain in general. In section 3 the interface language between coach and players will be described. The ORCA implementation of a coach will be discussed in section 4, section 5 discusses the first coach competition, and finally section 6 concludes.

### **7.2 The online coach in RoboCup**

The online coach capabilities are restricted to observe the game and communicate with the players [6]. Nevertheless it is a useful tool to improve the overall team performance [5, 17]. The coach receives global and noise-free visual information about all movable objects from the server. This makes coaches a valuable tool for game analysis and opponent-modelling, because it can communicate advice and information to its players. To prevent coaches from micro-controlling players and thus spoiling the distributed multi-agent character of the simulation league, its communication is somewhat restricted. It can send arbitrary free-form messages only during breaks in the game. Since 2001 it can also send messages in a standard language in certain intervals during play-on mode. An overview of this language is given in the next section.

### **7.3 The standard coach language**

The standard coach language enables coaches and teams that were designed by different research groups to work together. Because of this it is even possible for a research group to focus completely on implementing an online coach without having to put up with creating a team.

The language consists of five message type where two of them contain most of the semantic power. These so-called info- and advice-messages are basically rules that describe the observed behavior of a team or advices about how to behave respectively. The syntax of info- and advice-messages looks like this:

*(info TOKEN<sub>1</sub> TOKEN<sub>2</sub>...TOKEN<sub>n</sub>)*

and

*(advice TOKEN<sub>1</sub> TOKEN<sub>2</sub>...TOKEN<sub>n</sub>)*

The tokens in both message type have exactly the same syntax:

*(TTL CONDITION DIRECTIVE<sub>1</sub> DIRECTIVE<sub>2</sub>...DIRECTIVE<sub>n</sub>)*

TTL denotes the Time-To-Live which specifies how long a message should remain valid. CONDITION is a boolean expression constructed of predicate-primitives and denotes situations in which the DIRECTIVES are active. DIRECTIVES finally contain info or advice about actions that a team, a set of players, or a single player do or should do respectively.

In the case of advice the players can consider the rules within their decision modules and decide whether they follow the coach advice or rather follow their own behavior. An example:

```
(advice
  (6000
    (and
      (bowner opp {0})
      (bpos
        (quad
          (pt 40.0 15.0)
          (pt 52.5 15.0)
          (pt 52.5 -15.0)
          (pt 40.0 -15.0)
        )
      )
    )
    (do our {5} (mark {11}))
  )
)
```

This advice suggests the following: When any player of the opponent team (0 denotes all players) owns the ball and the ball's position is in a certain rectangle in front of the goal, the player with the uniform number 5 is advised to mark the opponent player 11.

In the case of info the players can use the rules that describe player behavior to make their own inferences. Another example:

```
(info
  (6000
    (playm ko_opp)
    (do opp {9} (bto {10}))
  )
)
```

This message informs the players that the opponent team has the tendency to execute their kick-offs by letting player 9 pass to opponent 10 (bto means ball-to). The players can use this information to update their tactics appropriately. E.g. they can try to mark opponent player 10 or focus their attention on intercepting the ball on its way from one opponent to the other.

Of course, to handle messages from the coach the team designers have to spend some thought in their players' behavior and decision modules. Especially when designing a team that can be used with coaches of other research groups, the behavior has to be very flexible. The ORCA approach on this matter is described in chapter 8.

## **7.4 The ORCA online coach**

### **7.4.1 General approach**

The ORCA online coach takes advantage of the many analysis methods provided by the TimeSlice-class (described in section 6.2). As described before, the offline analysis tool fed information from logfiles into the TimeSlices. The online coach does almost exactly the same, since the visual information that it receives during the game is very similar to that in the logfiles. Thus, during the game the coach maintains a lot of analyzed data. The main concern is to produce advice from this data that will be useful for the team.

Most of the methods described in chapter 6.1 take a lot of computation time and need many instances of data. But the online coach is required to come up with exploitable observations very soon to maximize its pay-off from the beginning of the game. Also, in case that the opponent team changes its behavior, the coach has to create new advice after very few observations. To achieve this, the coach sends advice- and info-messages based on statistical data at fixed intervals throughout the game, recalculating its advice for every communication. Note that as of now the time needed for recomputation is insignificant since much of the work is done by the TimeSlice-method in each cycle. So, after the fixed intervals the coach just has to pick up the analyzed data.

The different methods to provide advice and information are described in the following.

### **7.4.2 Marking**

In defense situations efficient coordination between defenders is important. In a team whose defense relies on marking, not marking a forward at all or marking a forward with two defenders simultaneous is suboptimal behavior. Communication might help overcome these situations. Yet, in the RoboCup domain communication is time-consuming and unreliable. Another method is to agree on locker-room agreements [19]. But locker-room agreements are not adaptive and thus cannot handle information acquired during the game. So the online coach is the optimal tool to coordinate marking assignments.

The ORCA coach identifies opponent forwards and its own team's defenders. Since several teams use dynamic role exchange [13] the ORCA coach executes the identification procedure for every advice. Identification is based on the player's position during a certain, manually chosen, timespan. Since the numbers of forwards and defenders and the average positions vary from team to team (and even within a team depending

on its current tactics), we developed the following method. We assume that each team consists of three sets of players, defenders, midfielders, and forwards. Also we assume that the position differences within a set are less than the differences between different sets. This allows the coach to look for clusters and classify the players accordingly. Although it is obvious that not every team discriminates its players into these three sets and that the position differences are fluent, experimentation shows that the identification results match human intuitive classification.

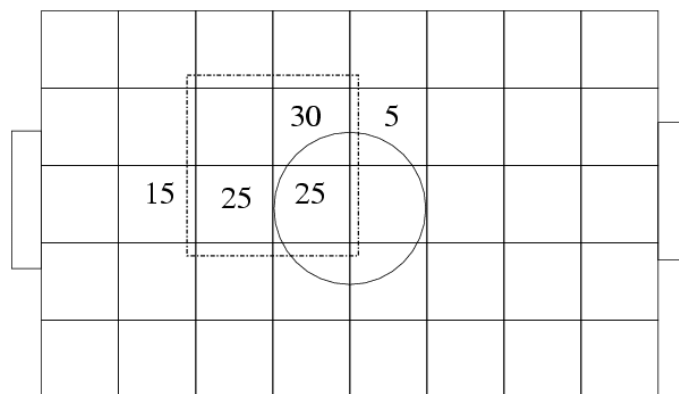
The next step is to assign defenders to forwards. A greedy algorithm based on spatial distance between defender and forward positions is used.

### 7.4.3 Defensive formations

Experiments showed that changing only parts of a team strategy results in inefficiencies [5]. For example, although defenders are assigned to the closest opponent forwards as described above, they tend to run long distances from their homepositions to their assigned marking tasks. Team performance is better if the defensive formation is fine-tuned to match the marking assignments better. The ORCA online coach also suggests home positions to its defenders.

As mentioned above the coach analyzes the opponent forwards' positions in situations in which they attack. Each cycle the position of the opponent players is counted in a grid that overlays the field. In [16] a similar method is used to match teams to predefined opponent models. The ORCA coach modifies this method in order to determine a defense formation as a function of the opponent's offensive formation.

Since player positions depend heavily on the ball and other player positions, it is not trivial to determine an opponent player's likely position during an offensive situation. The resulting grid for a player might look like this after adding positions into the grid at each cycle.



----- Resulting formation rectangle for given player

The numbers denote the percentage of cycles in which the player was in the according grid section. Obviously the region that a player uses as a home or action region is very unclear. The ORCA coach does not consider every possible position but focusses on finding regions in which the player will be with a high probability. To facilitate implementation it is assumed that this region can be described with a rectangle. A greedy algorithm is used to find the smallest possible rectangle that covers grid sections that add up to a certain percentage threshold. This rectangle is considered as the player's offense region.

As of now the defenders are advised to position themselves somewhere in the offense region of the forward they are assigned to mark. This creates a spatial distribution of the defenders like a defensive formation. If the defenders follow the coach's advice, they are positioned near to the forward that they are assigned to mark. This reduces the ways that the defenders have to run in order to pursue their different tasks.

#### **7.4.4 Detecting opponent setplays and formations**

Online coaches not only have the capability to issue direct advice, but can also communicate information to their players. This provides even more research opportunity on opponent modelling.

The ORCA coach is designed to work with teams of different research groups. So it cannot rely on its players to handle arbitrary information in their decision processes. Therefore the ORCA coach focusses on providing positioning and formation information about the opponent.

The formations are a direct byproduct of the method that identifies opponent forwards and assigns nearby defenders to mark them (see section 7.4.2). For each opponent player there exists a spatial distribution grid. With the aforementioned algorithm compact rectangles are created for each player. This information is sent to the players so that they can incorporate opponent positions into their decisions.

We do not believe in identifying standard formations because player positions depend heavily on marking, ball movement and noise [16]. So, observing spatial distributions of actual player positions and communicating these to the players looks more promising, because opponent players are likely to decide similar on marking and movement in consecutive offense situations.

Another aspect where the ORCA coach provides opponent modelling information are opponent setplays. It can be observed that several teams use fixed positions to respond to the goalie kickoffs. The coach looks for repeating positions in standard situations and communicates them to the players if it found stable positions. If the players are able to use these information, they can move to free positions or mark opponents faster than if they had to rely on their own limited view.

### **7.5 Experiences drawn from the first coach competition at RoboCup 2001**

The first coach competition was held at RoboCup 2001. All participating teams provided a team and coach each, which supported large parts of the standard coach language. The tournament modus was that each team was coached by all coaches except its own. Winner was the coach who accumulated the most goals in its games. To our knowledge these were the first games in which teams and coaches from different research groups worked together. The overall result was that each team played worse with a foreign coach than in a baseline game in which it was not coached at all. Though, additional experiments by Patrick Riley and Gal Kaminka of the participating ChaMeleons/OWL team [17] revealed, that all coaches performed better than a coach that sent random advice. Still, the coach competition event had to be analyzed.

An important observation is that not even one of the teams supported the whole standard language. In most cases info-messages were ignored totally, so opponent-modelling information provided by the coaches was of no use at all. In one case these

info-messages were even interpreted as advice-messages due to a misunderstanding on human level. In one game these info-messages contained information about the formation of the opponent players, so the coached team ended up using a mirrored formation with the defenders in front of the opponent goal and the forwards in front of their own goal. Formations were problems anyhow, because the teams used different concepts of home positions. Not all of these concepts were consistent with the home-directive.

One team crashed as soon as the coach sent a `playm`-condition because of an error in the language documentation which claimed the proper keyword was `pmode`. Additional complications arose from the fact that the teams integrated coach advice in very different ways. One team always followed advice, others only sometimes, and another team generally ignored certain advice in some situations. So coach instructions that relied on fine-tuned advices like OWL's setplays [17] or the marking- and defense-formation of the Dirty Dozen coach were likely to fail, because some teams did not interpret these instructions as "all or none". The interpretation of the games is difficult even when analyzing the logfiles, because the detailed implementation and decision processes of foreign players are in large parts beyond our knowledge.

The overall lesson learned from this is that a standard language is only as good as the human designers agree on its semantics and interpretation. Also there is no use in a standard if only parts of it are implemented. Finally, when plugging together systems that were designed by different groups, a testing phase is indispensable.

We'd like to thank Patrick Riley from the ChaMeleon/OWL team [17], Yang Yang from the Wright Eagle team [9], and Omid Aladini from the Hella Respina team [8] who provided detailed information about the implementation of their teams and coaches. Without their help our experiences from this competition would not have been possible.

## 7.6 Conclusion

This chapter described the approach of the Dirty Dozen coach and the advice it provides. A method how to efficiently use this advice in players will be introduced in chapter 8. Online coaches provide the opportunity to focus on opponent-modelling, and the standard coach language encourages cooperation between different research groups by pairing team and coach. The coach is a powerful tool, because it can consider information that is not directly accessible to player agents. Yet integrating its advice is not trivial, especially when working with foreign teams. Using teams with coaches has high demands on the flexibility on both the team and the coach side. The coach on the one hand has to find out the team's flaws and needs and has to come up with useful advice. But on the other hand it also has to observe the effect of its advice to ensure that it does not distract the players more than it helps them, as happened during the first coach competition. The team has to be designed so that it can get maximal pay-off out of the information and advice that the coach sends.

# Chapter 8

## SFLS

### 8.1 Introduction

Specifying the behavior of a multi-agent team is not trivial and most of the time only possible for domain experts [12]. Often the tactics, strategies, and overall behavior are buried somewhere in the system, sometimes even distributed throughout many files of source code. In these scenarios modifications to the team are time-consuming, error-prone, and not transparent. Also, if the behavior is not explicitly represented but implicitly within lines of program code, automatic adaptation is very hard.

This chapter introduces a method called Strategy Formalization Language (SFL) which we implemented for the RoboCup domain. By representing the team behavior in SFL, humans can modify it easily and fast without having to recompile the source code. Even online modifications to the team strategy by a coach are possible.

The next section gives an overview of the language concepts of SFL, particularly by comparing it to Clang concepts. A section about the implementation of a SFL system follows, and the last section concludes.

### 8.2 Strategy Formalization Language - Concepts

SFL is based on the standard coach language (Clang) [6] (see section 7.3). One of the design concepts of SFL was to make it downwards-compatible with Clang in order to facilitate integration of advices by an online coach [5] (see chapter 7). Clang alone is not detailed enough to specify a team's complete behavior. SFL extends it by adding primitives to the set of conditions, actions, and the control keywords, and by abstracting several Clang concepts. These additions are described below.

#### 8.2.1 Abstracting Clang concepts

One observation in the RoboCup domain is that certain behaviors need to be executed in situations where the player's exact identity does not matter. For example often players need to intercept the ball after a pass. Each player needs a behavior to achieve this. But Clang only provides constant uniform numbers to denote players. SFL generalizes the uniform number concept by introducing variables and primitives for situation-specific symbols. We will discuss both of these in turn.

Uniform number variables (i.e. variables that denote an uniform number) are the only way to refer to the same player in different condition tokens, for example

```
(and
  (ppos opp {X} SOME_REGION)
  (ballinterceptable opp {X})
)
```

denotes the situation that an opponent player is in a certain region and is able to intercept the ball. But with variables it is not only possible to map this condition to true or false, but they are also the only way to denote the same player in the directive part:

```
(advice
  (6000
    (and
      (ppos opp {X} SOME_REGION)
      (ballinterceptable opp {X})
    )
    (do our {3} (mark {X}))
  )
)
```

Situation-specific symbols for uniform numbers denote players that obtain certain functions in different situations. It is a frequent situation in RoboCup that the player that is closest to the ball should intercept it. Constant uniform numbers that are used in Clang are of no use to express this. SFL extends the expressiveness of Clang by introducing primitives like "ClosestPlayerToBall" or "FastestPlayerToPlayer". The specific player number referred to will be bound to the symbol in the cycle in which the rule fires. This allows to formulate many micro-situations in a very concise way.

Another extension is the parameterization of existing concepts in Clang. The semantics of actions in Clang are very general. For example, "(pos REGION)" means that the player should position itself in a certain region. Region can refer to arbitrary portions of the field and to locations of players or the ball. So it is a huge difference if the player should return to its homeposition when it is under no explicit time pressure compared to the situation in which it has to hurry to an opponent player in order to mark it before the opponent can get the ball. SFL introduces a parameter to the position to denote the power that a player should spend for its moves. In conjunction with the stamina-condition (see section 8.2.3) this can be used to formulate stamina-saving tactics.

There are more actions that were extended by parameters. See the appendix for the whole grammar of SFL in Backus-Naur.

## 8.2.2 Control keywords

A very important feature of SFL is its capability to specify rules that cannot be overridden. In the first coach competition at RoboCup 2001 the ChaMeleons team [17] used hard-coded behaviors that could not be overridden by coach advice. This is a useful method to make sure e.g. that the player that is the fastest to the ball will intercept the ball. It is very easy to specify this in SFL using the "force" keyword. A rule that contains this keyword will be executed no matter how many more rules are active. If there are more than one "forced" rules, only the first one will be encountered, because then

the matching process terminates. This can also be used to speed up the rule evaluation process, similar to the cut in Prolog.

### 8.2.3 Conditions

The aforementioned introduction of uniform number variables and situation-specific symbols already extends the condition specification expressiveness. But in order to implement a team more low-level and high-level concepts are needed. These concepts are introduced by adding primitives to Clang. First some of the low-level concepts will be described, followed by some high-level predicates.

In order to determine if a player can get the ball before any opponent does (it does not need to be the closest player to the ball, cf. section 8.2.1), the predicate (*ballinterceptable TEAM UNUM\_SET*) is introduced. It is true, if any player in UNUM\_SET of the given team can get to the ball before it moves out of bounds or is controlled by an opponent player. Depending on this condition e.g. offensive or defensive actions can be executed.

Another low-level predicate is (*ballvelocity VALUE*) which checks the velocity of the ball. VALUE can be a constant or a variable. Variables can be used in conjunction with the less-, greater-, equal-predicates that are also added in SFL. For example, the power that a player should exert in order to get the ball can be specified depending on the speed of the moving ball and (also introduced in SFL) the player's remaining stamina.

Several teams change their tactics and formations based on the goal difference and remaining time [19]. SFL contains high-level predicates like these to specify the behavior based on these strategically aspects. See the appendix D for the whole SFL syntax.

### 8.2.4 Actions

Some of the actions of Clang have been extended by parameters (see section 8.2.1). The set of Clang-actions is quite exhaustive. SFL introduces only one major concept. *Interceptball* causes the player to get to the ball as fast as possible.

## 8.3 Implementing SFL

In this section an implementation of SFL will be described, the Strategy Formalization Language System (SFLS). It should be noted that this is only one of the different ways to implement a multi-agent system using SFL. The system described here consists of several modules: the parser which builds objects for each rule, the matcher which evaluates which rules are active at each time step, the selector which decides which one of the active rules should be executed, and finally the effector which decomposes the selected actions into server primitives and executes them. Each of these modules will be described in more detail in the following.

### 8.3.1 The parser

The soccerserver package contains a lex/yacc parser for parsing the standard coach language Clang that is used by the soccerserver itself to recognize legal coach-messages. The Clang parser translates Clang messages into C++ objects by creating a new object

for every message and every non-atomar element of the message. The result is a hierarchical object-structure representing the message. Developers are encouraged to use this parser for coach-message parsing in their agents.

lex/yacc is a set of parser-generator tools that provides a syntax for describing a grammar by specifying lexical entries and production rules and from this description generates a C-program that is able to translate strings from the language generated by that grammar into any data structure. For that purpose, grammar rules can have pieces of C-code attached to them that are built into the generated parser. This code is executed when the rule is applied during the parsing process and is used by the generated parser to build up the output data structure using the input of the applied grammar rule.

As SFL is an extension of Clang, implementing the parser simply consisted of extending the Clang parser by adding lexical entries and grammar rules for the concepts new to SFL and of providing the classes in whose instances they are to be stored.

The parser's input comes from the initial behavior-file in the form of SFL-rules read in at the start of each agent's lifetime, and, via the server, from the coach client in the form of coach messages during the game. After a rule is parsed, its object representation is stored in a rulebase, where it remains until its time-to-live has expired. The modules matcher, selector and effector work with SFL-rule objects rather than with SFLS-rule strings, so that parsing only has to be done once for each message arriving.

### 8.3.2 The matcher

At each time step the matcher determines which rules are active. This is done by evaluating the condition parts based on the world model of the agent. Rules whose conditions are evaluated as true need also be checked if their directive parts refer to the agent. Only in this case the rule will be handed over to the selector module which will decide which of the active rules should be executed.

#### Variable- and symbol-handling

Evaluating conditions is called matching, because similar to Prolog it tries to prove a condition based on the current world model. Some conditions are checked straightforward, like the play-mode condition. But several conditions can contain variables. SFL uses two types of variables:

- uniform number variables whose domain is  $0,1,2,\dots,11$
- real number variables which denote an integer or float value like the time cycle or the speed of the ball.

At the beginning of the matching process all variables are uninstantiated. When encountering such an uninstantiated variable, the matcher assigns values which are derived from the world model. In the Dirty Dozen world model all variables in SFL can be instantiated as soon as they occur. Handling of real number variables is easy. A variable is either instantiated or not. There is no concept in SFL that can fail when using ungrounded variables, so assigning values to uninstantiated variables will always result in an evaluation as `true`. Beginning at the second encountering, real number variables can make conditions fail. Uniform number variables are a different case. They are handled similar to domains in Constraint Satisfaction Problems [11]. That is, these variables represent sets of uniform numbers that satisfy the condition. These sets are reduced by consecutive conditions. An example might illustrate this.

Let us assume we have the following SFL condition:

(and (ppos our {X} REGION\_A) (stamina our {X} high))

Let us also assume that in the current situation there are the players with the uniform numbers 2,3, and 4 in REGION\_A, where only player 3 has a high stamina level. Player 5 has also a high stamina level, but is outside of REGION\_A. So, when encountering X for the first time, the matcher will instantiate X with {2,3,4}. In the stamina condition it needs to cut the domain of X by removing 2 and 4, which do not satisfy this condition. If not even 3 had a high stamina level, the whole condition would fail. If the conditions were connected by an `or`-junction, the set that satisfies the `ppos`-condition had to be unioned with the set that satisfies the `stamina`-condition, resulting in {2,3,4,5}. The situation gets more difficult, if negations are used with nested conditions. So each condition needs to be evaluated considering its context with negations and junctions.

Situation-specific symbols like `ClosestPlayerToBall` have to be evaluated at each time step, too. They evaluate to exactly one uniform number so they can be treated like constants afterwards.

Rules that contain uniform number variables or situation-specific symbols in their directive part have to be evaluated before the matcher can determine whether they refer to the agent. If the agent's uniform number does not appear in the constant uniform set of the directive part of a rule, the matcher can skip this rule, because the actions do not refer to the agent.

The values of variables have to be stored longer than just for the rule evaluation, because the selector has to work with them.

## Definitions

Just like in Clang, in SFL it is possible to define conditions, regions, directives, and actions, in order to refer to them by a short handle. The matcher also manages these definitions by maintaining a table of names for each class of definitions. Basically, the defined concepts are stored as objects just like the other rule components and linked into the matching process if their names are encountered.

### 8.3.3 The selector

As mentioned before the selector chooses the best rule from the active rules. In this implementation this was done in a simple, yet effective way. Each rule is assigned a fixed priority. The basic idea behind this is that rules are heuristically evaluated on each of the three Clang levels (actions, directives and conditions), being assigned three fitness values that are summed up. So, certain actions seem more promising, e.g. `interceptball` has a higher fitness than marking on the action-level. Directives refer to different sets of players and the more specific a player set, the higher the directive's fitness on the directives level. E.g. the fitness of a directive that refers to the whole team has less fitness than a subset, which again has less fitness than a situation-specific player-symbol.

Since the rules specified in the team-implementation are fixed, the fitness-assignment is done manually, but automated assignment will be straight-forward.

The third level is based on the conditions and is basically a way to save world knowledge from the rules and incorporate them into the selector. So it is not necessary to specify in a rule that the agent should only mark an opponent if no teammate is already there. Certain common sense heuristics can be used on this level to assign a fitness value to each rule. This has not yet been implemented, so the SFL-rules in our team contain certain amounts of this common sense knowledge explicitly.

The selector will then execute the rule with the highest fitness, unless one of the rules contains the "force"-flag which denotes that this rule should always be executed if its condition is true.

The action is then handed over to the effector-module.

### 8.3.4 The effector

In our SFLS-implementation the effector is a straight-forward mapping from the directives, that the selector provides, to the low-level skills. Thanks to the CMU code there is a large set of low-level skills and functions. So the effector basically does some checks whether the given action can be executed directly or needs some more decomposing.

SFL does not require to list all necessary conditions for an action in the first place. Although some actions might already been filtered out (in future version of the selector module) if their conditions are not satisfied, the effector for example still checks if the agent is close enough to the ball and facing the correct direction before executing a pass.

As of now, if an action fails and cannot be executed, the agent will do nothing in the current cycle. In later versions the effector should be able to request another action from the selector, if there were several to choose from. For example, if a pass-action contains a whole set of uniform numbers, the selector will only pass one to the effector. If for any reason the action turns out to be impossible, the selector needs to provide one of the other uniform numbers in the set as a target.

### 8.3.5 Integrating coach advice

In our SFLS-implementation integrating coach advice is straight-forward. Each advice token that the coach sends during the game is added to the rule base. The priority of coach rules is a high fixed value. This way it can be guaranteed that initial rules that the team designers do not want to be overwritten by the coach can be assigned a higher priority, retaining the possibility that default or less important rules can be overwritten by coach advice.

This simple method is successful as experiments [18] with our team and foreign coaches show. These experiments were made at the Carnegie Mellon University and revealed that their coach can significantly improve the score of our SFLS-team. Since the changes made by the coach only affect the rulebase, this also proves that our approach of declarative agent-modelling is promising and that the performance of our team can still be improved by specifying new rules.

## 8.4 Conclusion

Behavior specified in SFL is easily and fast modifiable. Also incorporating coach advice is possible and leads to successful results. As experiments by CMU showed, the performance of our team is highly flexible and depends on the rules in the rulebase. So tweaking these rules should improve our team in the future.

Compared to Clang, SFL is more expressive and rules can be formalized more concise. While we are positive that the language SFL covers anything that a team needs, the implementation of a system that interprets SFL still offers more research opportunities. Based on the observation that in both Clang and SFL there are more condition-

than action-primitives, we believe that in soccer the knowledge when and why to execute certain actions is crucial. Therefore, more work should be done in the selector module in order to decide more dynamically which rules are executed. This also includes handling coach advice more efficiently, since it is integrated with a manually fixed priority now, and backtracking of actions if the effector reports that an action is not possible.

SFL is just one way to formalize strategic behavior. There is no general agreement on what a strategy is and how it should be specified. The related work of COACH\_UNI\_LANG [14] should be pointed out which formalizes strategies in terms of roles, formations and tactics. Unlike SFL which is based on situation-action mapping, it uses player types by setting parameters. Thus, its notion of strategy is different than that of SFL, which uses it synonymous to behavior.

## Chapter 9

# Testing, Debugging and Tuning

### 9.1 The Gauntlet

#### 9.1.1 The purpose of a gauntlet

During the development of our RoboCup team we encountered the problem of evaluating changes in the code. A developer (or group of developers) would implement a new feature, redesign a strategy or fix a bug and then test the new team at his home computer or on a couple of computers in the campus. The problem was that the results we got from different groups were not comparable. On some machines we would lose to team A. On another machine we would win. Another problem were the random factors which are induced into a RoboCup game by the soccer server. Even if you replay a game in exactly the same configuration (code and hardware) you get significantly different results. To even this out you have to run each game a couple of times and then work with the average outcome.

So we implemented a weekly (and later nightly), automatically run tournament (gauntlet in our lingo). In this tournament we took the latest release of our team from the repository and set it up against a number of other RoboCup teams which were available on the internet. We had different configurations of our team and each one of these configurations had to play against every other team. This resulted in a number of pairings and each pairing was then run five times to try to reduce some of the randomness. Each game was logged by the logging mechanism implemented in the soccer server. These logfiles were saved and processed. We created visualizations of important facts about each game and set up a number of HTML-pages which showed the results.<sup>1</sup>

#### 9.1.2 Design criteria

First we needed a way to start games remote. That means that we needed a program which is run on machine A, starts a server on machine B, a team on machines C and D and then tells the server to do a kickoff. This program would be used for the gauntlets but also had it's uses outside of them. If a developer wanted to see a game he/she didn't have to dabble around with three different telnet-sessions. Another goal was to have the program transfer the team- or server-binaries automatically to the machine on

---

<sup>1</sup>Of course this was all done automatically.

which they should run. But this should only happen if the local version is newer than the existing remote one (to reduce traffic).

The main gauntlet program then just needed to start a remote game five times for each pairing it came up with. The pairings were computed automatically by the gauntlet program by scanning a directory for subdirectories containing team-binaries. This had the advantage, that if one wanted to change the teams of a gauntlet, one had only to add (or remove) a new team directory. Everything else would be deduced automatically.

Another design constraint was that we had to get information from the teams and the server back to the gauntlet program. From the server we got information about the coordinate of the ball in each time frame. We used this information to plot a graph using the X-axis as the timeline and the Y-axis to show the X-coordinate of the ball<sup>2</sup>. We also wanted information about the score and meta information on the game from the server. Meta information is stuff like „which team has the ball for how much of the game”, „how long does the goalie have the ball”.

To run the clients on the remote machine, we used a script to start the eleven (or twelve when using a coach) processes needed to run the team. Using a script had the added advantage of getting information about the state of the clients. We used this mechanism mainly to detect and report segmentation faults in our clients.

### 9.1.3 Implementation

The machines we wanted to use for our gauntlet were all reachable via `ssh`. We could thus use the feature of the standard `ssh` client to execute a program remote (just like `rexec`, only encrypted). The output the remote program was echoed back - it became the output of the `ssh`-client. So the server and the script to start the team didn't have to worry about sending their information across a network. They simply printed it to `STDOUT`.

To get the information from the server we implemented a trainer (or offline coach) which collected the events sent from the server. It then printed the relevant information (ball position, score and meta information) to `STDOUT`. The script that started the team worked similar. It parsed the output from individual players and wrote the relevant bits (no pun intended) to `STDOUT`.

All this data came together on the machine running the gauntlet (which was different to the machine running the server and to the machines running each team). There a logfile was created for each pairing (every logfile contained the data for five games). After the whole gauntlet was finished, the data was then visualized.

The visualization was done by a Perl-script which created a web page for each pairing. The page contained the results for the five games and five graphs depicting the above discussed curve. If there had been any segmentation faults in our team this information would be given as well. The Perl-script further created index pages to easily access the information-pages. Then all the freshly generated HTML-pages were uploaded to an internal webserver and could be accessed by all team members.

---

<sup>2</sup>This may sound confusing, but it just showed a curve indicated in which half (and how far in that half) the ball had been throughout the game

## ATTCMU2000 - BB\_Seattle1

Results:  
1:0  
3:0  
1:0  
3:0  
1:0

Segfaults:

Server logs: [logs/996544657.log.bz2](#)  
[logs/996545758.log.bz2](#)  
[logs/996546535.log.bz2](#)  
[logs/996547314.log.bz2](#)  
[logs/996548096.log.bz2](#)

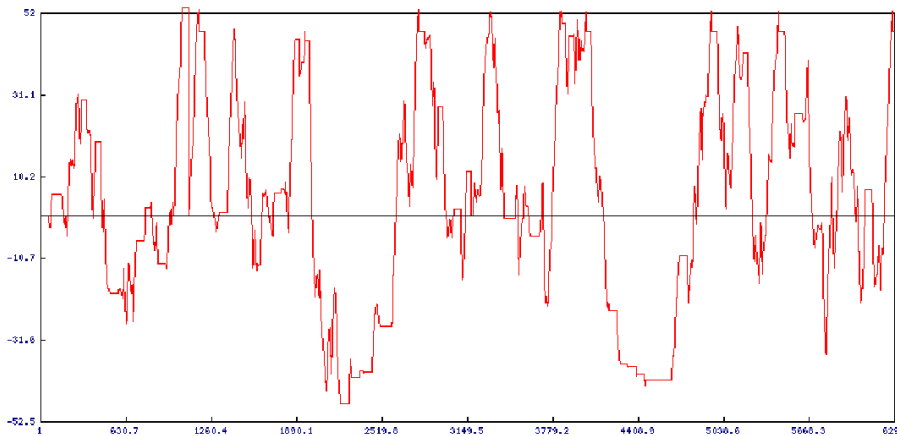
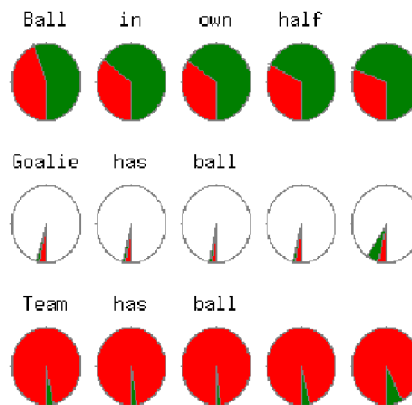


Figure 9.1: This is a screenshot showing the visualization of the first game of the pairing ATT CMU 2000 versus our team (Osna BallByters). This gauntlet took place in preparation for the 2001 World Championships in Seattle.

## 9.2 Quality Assurance Management

### 9.2.1 Introduction

At some point in a project it becomes apparent that there has to be some way of controlling the quality of the produced product. Especially if there are, say, more than two or three people working on a piece of software, the need to have one (or more) persons testing for bugs and / or logical mistakes soon arises. This section will be concerned with our experiences with 'Quality Assurance Management' (QAM).

The remainder of this section is organized as follows: I will give an overview of the tasks of quality assurance management and the general principles to adhere to first. Then there will be a section about the tools we used in our project. A section about our experiences and difficulties with quality assurance management follows, and last but not least an outlook on what we could have done better.

## 9.2.2 Tasks and general principles

The purpose of having someone to test a software product as a whole is twofold. First, there is someone whose task is explicitly to test and evaluate the software. As opposed to the individual programmer, who may only test his / her code, a quality assurance manager oversees the development of the whole code and the integration of new parts thereof. Only if new code has been tested for its integrity and sideeffects with the old code, will it be merged with the main branch of the developing project.

Second, the person responsible for QAM provides a point of reference for integrating new pieces of code in a ordered fashion (esp. concerning sequentiality). If there are version conflicts, e.g. when two programmers are working on the same file, QAM can make sure the involved parties are notified of the conflicts. That way they can decide together on the proper solution.

Merging the code of a group of people working on the same piece of software can become a time-consuming task. Obviously it is also not enough to code and test parts of a software product only, but the complete code has to be evaluated and tested for its integrity. It is also useful to have a standard procedure of integrating new code into the software. For these reasons the ORCA project decided on designating someone as the responsible for quality assurance management.

## 9.2.3 Tools and Procedures

A very useful tool for general version control of a software product, which we used extensively, is CVS [1]. CVS is described in section 9.3. CVS is capable of handling most of the version control by itself. Because it notifies the user of version conflicts it cannot resolve, QAM can enter at that point and coordinate the involved programmers.

It is also advisable to create different branches for the development version and the release version of the software. Changes from the development branch should only be subsequently incorporated into the main (release) branch by the person responsible for QAM. That way the developers can try out different approaches without changing the stable release. QAM can then ensure that each change is tested and evaluated before being integrated into the release. This is especially true for incorporating multiple changes to different parts of the code, when it matters most to provide for sequentiality because of possible side-effects.

We also decided to create a testing environment similar to the actual competition situation by setting up three designated PCs running Red Hat Linux 6.2. We have made the experience that although it is possible to run both teams, server and monitor on the same PC, results may vary greatly from game to game and differ by quite a margin from the actual results in an environment where teams and server are distributed on different computers. We also used this setup for further testing as described in the gauntlet section of this document (see section 9.1).

Another decision we made was to extract certain parameters from the actual code and to include them in a file called 'orca.conf', similar to the Configuration files included in the SoccerServer or the CMU-Code. That way it was easier to modify those parameters and to test their effects in the gauntlets.

## 9.2.4 Experiences with QAM

This section will describe the way we actually worked with quality assurance management. It turned out to be a sometimes rather awkward task, due to inexperience, and a

sometimes very fun task, due to seeing promising improvements in agent behavior.

### **Actual day-to-day handling**

When deciding to use a software like CVS to handle version control, it is advisable to make sure everyone involved knows how to handle the system. This is especially true with respect to the Update-Code-Update-Commit Cycle as required by CVS. Even CVS is only as good as its users, and forgetting to update e.g. *before* starting to work sometimes led to ‘forgotten’ lines of code. So did not remembering to commit all the files changed. It is of course possible to commit all files at once, but sometimes we didn’t want that for reasons of e.g. different comments or not intending to commit certain changes because they had turned out to decrease performance. When working with multiple branches on the command line, extreme care needed to be taken when changing branches quite often in a single work session.

Organizing sequential check-ins is also one of the tasks of quality assurance management. Especially when it came to deadlines (like tournaments), multiple last-minute changes tended to somewhat evade thorough testing. They were often integrated quite fast, without proper testing between the integration of the different new pieces of code.

### **Evaluation of games**

In our project, QAM also became responsible for preliminary evaluation of the results of the games played. Whether a feature improved agent performance or not was then usually decided on a broader basis, meaning usually a decision by all the programmers working in that area. So QAM is primarily responsible for the quality of the code, not the performance quality. Of course it is virtually impossible for a single person to know about all the code of a piece of software. Therefore, certain more general parts of our agents (e.g. the SFL team, see section 8) were left under the supervision of their respective programmers.

We used two methods to evaluate the behavior of our agents. The first, and obvious one, is to watch games. We used the FC Portugal 2000 team [2, 13] as our default opponent. With the setup described earlier it was possible to watch the behavior of our agents under quite ‘realistic’ conditions. Secondly we used weekly and daily gauntlets (as described in 9.1) to gain an overview of team performance against different teams, also with different configurations files. This was useful to have a broader basis of game results for decisions concerning further needs for improvement.

### **Branching**

Using the branching capabilities of CVS helped us a lot. The SFL part of the code was mainly developed in its own branch and later merged with the main branch. For the Playtree-Version of the DirtyDozen-Team different branches for offense, defense, and communication were used at different stages of the project. New approaches were implemented and tested in those branches before being considered for the main branch. If branches hadn’t been used for some time it happened that merging code turned out to be comparatively time-consuming.

### **Shortcomings**

We didn’t use branching as extensively as we perhaps should have. We should definitely have had a separate release branch in addition to our main development branch.

This release branch should, in the best case, only be used by the person responsible for QAM, and very cautiously at that.

Paying more attention to keeping branches up-to-date would certainly have saved a lot of time spent on merging them later. We had problems with the general self-discipline of committing changed code to the repository. Especially with deadlines looming, people tended to commit ‘improvements’ on a rather arbitrary basis. Introducing a test-bed earlier than we did would have been, in retrospective, a good thing to do. The same holds true for the gauntlets.

We also focussed too much on testing against FC Portugal 2000 because they were the strongest opponent and qualification opponent. Because of the superior abilities of their agents we were mostly concerned with improving our defense and neglected the offense. We developed offensive concepts and weren’t able to evaluate them properly because during games our team spent most of its time in the defensive.

### **9.2.5 Conclusion**

Ensuring the quality of the software produced is an obvious demand for any serious programming. In a domain like RoboCup where performance is measured in a relatively straightforward way, it is clear that the performance quality of the code is usually put first. This doesn’t eliminate the need to ensure the integrity and coherence of the actual code. Agents preferably run without crashing or using too much of the system and network resources. It is therefore useful to introduce a formal way of quality assurance. This can be done by designating someone to perform the described tasks. We found that having a quality assurance manager was definitely helpful.

## **9.3 CVS**

In the ORCA-Project CVS is used for version management. It is a tool to keep an eye on different versions of each project file. It can be used to merge different versions of the same file or to extract a patch file of the different versions of the same file. You can find a manual and other resources under <http://www.cvshome.org>.

In our project we have made good experiences with CVS, because we often had the problem that the actual version of our project did not work and with CVS we could roll back to a working version. This also helped us to locate a bug within 1 or 2 files most of the time. CVS also helped us to tag special versions and to split the development tree to have different branches that were important to be implemented, but lead to a not properly working version in the meantime.

Nowadays you can’t maintain such a huge project without using version management anymore. Even though we did not test other version management tools like velvet rose, CVS fitted to our needs, because it was available for free, you could get different GUI’s for it, it is quite easy to use, stable, and available for Linux.

# Chapter 10

## Tourneys

**“My father learned me once that making mistakes is not very clever.”**

While this is true it is not always fatal to make mistakes. Making mistakes and noticing you did so lets you grow. All people involved in this project have never participated in a RoboCup event. Thus making mistakes was to be expected. This chapter is about what we learned from the “mistakes” we made.

### 10.1 RoboCup German Open 2001 in Paderborn

One major point, that became obvious immediately, was that we never saw a simulation match the way it should look. The setup in Paderborn included a well running network of Pentium-III with 800Mhz to use for the simulation competition. With several of these machines for every team everything went smooth. Up till then we had never seen a game running at the speed it should be running at. Our computers at home just weren't fast enough and the network at the university is of course not designed for and not exclusive to RoboCup simulation matches. Of course we had noticed that something was not running right, but the most shocking experience was that speed matters a lot in the simulation league. Even our team played a lot better under these conditions. The setback was that it didn't improve nearly as much with these extra resources as the opponents did.

To build a good team in RoboCup simulation requires a well tuned system to test it on. You need the best conditions for both your own team and the opponent. Without it you will not get valid results. The simulation league is highly timecritical. A couple of milliseconds here and there can drastically change the performance of single agents and the whole team.

Our experience in Paderborn showed us how important it is to coordinate a group of individuals to achieve a given task. We were making last second changes in parallel. That is in itself not a bad idea, but it includes the responsibility of communicating. It is terribly important that the person that is starting the team knows which build to start. By the time the contest started we had several different versions of our team. Each with different brand new and older features. Most of the newer ones were of course not thoroughly tested if they were tested at all. We thought about this matter beforehand and had a stable version to fall back to, but there was a major bug that wouldn't allow our agents to score. This bug had to be fixed and it was. That of course meant that our “stable” version was not on our CD but modified right there on the contest computers.

Several of us were hacking away right next to each other. Everybody was very absorbed in whatever he or she was doing so nobody really knew what sort of version the others were working with or on. That way some important and some minor bugs were fixed but nobody knew of all of them and especially if a certain version fixed all of them. This led to a lot of confusion and of course stress.

We also had different starting scripts. One of them made especially for the contest and an older one. The older one didn't include the file which held all of the important parameters. Of course in our first match due to our confusion we used the wrong script. The result being our players staying in their fixed formation and not going to the ball even if it was just a few meters away. Major malfunctioning like this results in big stress. It was our first tournament match and we didn't lose because we coded wrong, we lost because what we coded wasn't even being executed. An experienced team would immediately work on some measure that keeps this from happening. Instead of being the coolheaded professionals we wish we were, we started to freak out and turned on each other.

For a team in a hostile environment provided by a competition it is of unrivaled importance that the team members can rely on each other and support each other. There are always conflicts even conflicts of personal nature. These conflicts have to be resolved or put aside when it comes to getting the job done as a team. Talking about a hostile environment is not really appropriate when it comes to RoboCup competitions. One important thing we learned in Paderborn was that this really is a community of people working on problems in the same area. The competition is of course important and competitive. If you lose you lose and you're out of the competition. That doesn't mean that you're out of the community though.

## **10.2 RoboCup 2001 Seattle**

The second time we got into contact with the community was when we attended the Worldchampionship. Although Paderborn was international it was mainly Europe. In Seattle there was Europe, America, Asia and Australia. And maybe more important than the variety was the number of people that attended this event. There were just so many of them. Everybody was involved in one of the leagues. There were 44 simulation teams which meant you had enough people to get to know without ever talking to anyone from another league. There was of course some interaction between leagues but for the main part the leagues stuck to themselves. There were enough internal problems to solve.

It was not just many and different people. It was also important people. People you know by names from papers or because there on the committee. At first you stand there and stare in awe at these important people, but after a while you are getting problems with the system and you have to go talk to them. The big surprise is that they treat you as an equal. They treat you as somebody that is advancing the RoboCup community. This is an experience that cannot be made without going to gatherings as this one. Cutting edge researchers and scientists right there in the same room, the same competition and unbelievably socially on the same level as you are. By saying they treat you as equals does not mean that they talk to you as if you had gone through the same extend of learning, research and RoboCup experience as they have and indulging in out of this world terminology, but rather that they talk to you as someone who is interested in the same things you are. They are happy that you are there, that there are more people like them. Trying to achieve new things in RoboCup.

One of the important people was Gal Kaminka a member of the organization committee. He held a really good talk about what it means to do science. He stressed the theme of this years RoboCup: “Fun competition. Great Science. (tm)” One important point that he tried to bring across was the difference between doing research and doing science. If you find a solution to a problem and try it and it works very well, but you don’t tell anyone how you did it, you are a researcher but no scientist. An important part of science is telling people about what you are doing. If you come up with something that you think is new, check the literature. The chances are incredibly high that somebody already tried something very similar. If you can’t find anything ask people working in that field and they will point you to the literature. Gal really stressed this point (“There is always literature!”). Take a good look at what the other people have been doing and what they found out. Compare your results to theirs and show what is different. If something you do doesn’t work out the way you expect it to, try to find out why it didn’t work that way. If something doesn’t work at all try to find out why and most important tell people whatever you find out. If you do that you are doing science. In science it is important to tell people.

# Chapter 11

## Conclusion

### 11.1 Achievements

The overall success of the ORCA-project is obvious. Despite the fact that none of the student members had experience with a project of this size or even implementing in C++, a running and complex multi-agent-system was developed. It took part in the German Open and RoboCup 2001, so the project's qualification is undeniable.

Student projects like this one not only aim at achieving the given project and individual goal, but also aim at improving and acquiring social skills that are needed for working in a project of a reasonable size. Lessons were learned in project handling, e.g. time schedules, conflict management, presentations, and joint software development. Several methods for quality assurance and working plans that are used in company or academic projects were used. Also the student members got familiar with different scientific methods and tools. The experiences drawn from this project will be useful for future projects, not only for the ORCA members, but hopefully also for other student projects that might learn from our experiences described in this documentation in order to avoid the most prominent pitfalls.

Although this student project was planned to last only one year, there are plans to continue it. Throughout this paper several challenges that can be worked on were described.

### 11.2 Acknowledgments

During the last year we had a lot of fun and had the opportunity to really get involved in the RoboCup community. In order to be able to do this we needed a lot of resources. Fortunately, we received a lot of help. The ORCA project would therefore thank the following:

- the *Universitätsgesellschaft Osnabrück* for their support and search for sponsors. They made it much easier for us to afford the trips to Paderborn and Seattle.
- the *Institut für Semantische Informationsverarbeitung* not only for helping us to go to and stay in Paderborn
- the *Verkehrsverein Stadt und Land Osnabrück, e.V.* for the Seattle support

- the *Studentenwerk Osnabrück* not only for the shirts they gave us.
- the students of the University of Osnabrueck represented through the *student parliament* for helping us go to Seattle
- the *Binding Brauerei* for sponsoring us

But this all would not have been possible if it hadn't been for the two people that supported us the most:

- Prof. Dr. Claus Rollinger
- Wilfried Teiken

Their continuous help and encouragement was by far more important to us than any monetary support could have been.

# Appendix A

## Debug-API

### A.1 Introduction

For debugging purposes we included a Debug-API which you may use when working with our code. We give a short description on how it works.

### A.2 Basics

The Debug-API is defined in the `utils.C` and `utils.h` files. Especially `utils.h` is important because it holds some important definitions.

If you want to debug something you can use the `MAKELOG` macro:

```
MAKELOG((debug_level, debug_facility, message))
```

Depending on whether the debug flag is given at compiling time (`-DDEBUG`) the macro will expand to a debugging call. `MAKELOG` needs three parameters:

1. `debug_level` a number between 0 and 99  
`debug_level` sets a level at which the message will be considered for output with 0 being very important and 99 being least important. Through the levels you can control the amount of information that is suppose to be put out.
2. `debug_facility` describes a binary value  
the facility is the general group of debug information. In `utils.h` we included some facilities already.
3. `message` the debug string  
the message is constructed similar to strings for `printf`

When the code is compiled with the debug flag set the client may be started with debug options. If no options are given or if some are left out the default values will be used:

- `debug-fac` the debug facility as a binary number or a string  
default value: `DBG_ANY`  
if the binary number is in the set facility all entries with that facility will be shown. Possible strings are

<b>facility</b>	<b>binary value</b>
DBG_CMU	1
DBG_OFFENSE	2
DBG_DEFENSE	4
DBG_GOALIE	8
DBG_COMM	16
DBG_COACH	32
DBG_TRAINER	64
DBG_FORM	128
DBG_SFL	256
DBG_OTHER	512
DBG_ANY	2<<30
DBG_ALL	0

Table A.1: debug facilities

If more than one facility is needed the sum of the facilities needs to be entered. `DBG_ANY` will be used if no facility is given when the client is started. `DBG_ALL` will always be shown. Those two kinds of messages will get through if no debug facility is handed to the client when started. If you include a debug facility in your start script then only `DGB_ALL` messages and your chosen messages will be included. `DBG_ANY` will not be seen anymore.

- `debug-lev` the level as an integer from 0 to 99  
default value: 99  
if a given debug entry has a matching facility it is tested if the debug level of that entry is equal or lower than the set debug level. With the default value all messages should get through.
- `debug-file` a filename  
default value: `STDOUT`  
if a filename is given the debug information is stored in the file. Usually, the messages are send to `STDOUT`.

## A.3 Examples

### from our code

Since we used the Debug-API we include some examples that will help you understand how it works:

In `Memory.C` line 135 there is a debug line:

```
MAKELOG((40, DBG_OTHER, "adding %d Tokens:" ,tokens.size()));
```

It is a debug message that is not categorized and therefore the message is put into the 'other messages'.

In line 197 of the same document it states.

```
MAKELOG((30,DBG_OTHER,"Warning, named directives not yet supported."));
```

and in `MemFormation.C` it reads in line 353. Since it is a formation debug message, it is labeled `'DBG_FORM'`.

```
MAKELOG((1, DBG_FORM, "current formation: %s \n",
        currentFormation->name));
```

These will be used as an example to explain the general debug procedure.

### printing `DBG_OTHER` messages

If the code has been compiled with the debug flag and if the client has been started with `-debug-fac=DBG_OTHER` then the client will put out the line `'adding ...'`, and `'Warning, ...'` but not the information about the current formation since the last message belongs to a different facility<sup>1</sup>.

### printing all example messages

If the `DBG_OTHER` as well as the `DBG_FORM` messages are needed a different facility has to be set when starting the client. To get the new facility number the two values of the chosen facilities have to be added. According to the table A.1 above and the definitions in `utils.h` the value for `DBG_OTHER` is 512 and the value for `DBG_FORM` is 128. Therefore the new facility would be 640 and is set by starting the client with `-debug-fac=640`.

### refining output

Since the default level is 99 all messages in the example will appear since their level is lower than 99. But if only high priority messages are supposed to be considered than adjusting the debug level the client uses can help. The line `-debug-fac=640 -debug-level=30` will leave the `'adding...'` line from the first debug example line untouched since it's level is above the new debug level.

### separate filename

Finally, if the messages are supposed to be stored in a separate file instead of `STDOUT` the line `debug-file=filename` has to be added when starting the client.

## A.4 Known Problems

As we were working with the `Debug-API` we discovered that it didn't handle objects too well. In order to get the values of any given object each value has to be put in a string variable. So this seems to be not as comfortable as `cout << my_object << endl;`.

---

<sup>1</sup>Of course, all `DBG_ALL` messages will appear as well

# Appendix B

## Terms

In this section we will explain some terms that we use throughout this document:

- **basic skills**  
actions sent to the server; right now, the ORCA client uses the CMU skills as it's basic skill.
- **CMU**  
Carnegie Melon University; developer of the agent we used to base our team upon
- **Dirty Dozen**  
team name used during the Worldchampionship in Seattle
- **high level skills**  
combinations of basic skills; An action *sore goal* would be considered a high level skill since it involves multiple basic skill action like *kick*, *dash* or *turn*
- **logfile analyzer**  
can analyze games and gather information from log files. Since it doesn't follow a game in progress it has more time at it's disposal to reach it's conclusions.
- **offline coach**  
see logfile analyzer
- **online coach**  
interacts through the standard coach language Clang with teams. The Dirty Dozen team was developed to be highly compatible with the information coming from an online coach.
- **Osna BallByters**  
team name used during the German Open in Paderborn
- **positional terms**  
here are some terms we used to describe the player's positions. The values of those points are read from the file *formation.conf*
  - **HomePos**  
a point that describes the player's initial position. It is the center of all other positional values.

- **HomeRange**  
a circle in which the player is free to position itself.
- **MaxRange**  
a circle that is used to describe the maximum radius a player should take into account to calculate it's actions from
- **SFL and SFLS**  
one of our team's core elements is SFL and the SFLS components:
  - **Clang**  
the standard coach language that was agreed upon by the RoboCup community
  - **effector**  
converts the selected rule to a server conform action
  - **matcher**  
determines which rules are active at each cycle depending on the world-state
  - **rule base**  
a file that holds the different SFLS rules. It is read when the client is started. Throughout the game an online coach may add new rules to that base.
  - **selector**  
chooses a rule out of the active rules that were found by the matcher
  - **SFL**  
Strategic Formalization Language; an extension to the Standard Coach Language (Clang) that is used by our team to describe team strategies
  - **SFLS**  
A system that runs strategies that were specified in SFL. An example of such a system is implemented in the 'SFLS ORCA' team.

# Appendix C

## SFLS Rule Writing

In this section examples and methods on how to write SFLS rules shall be given. Please refer to chapter 8 for concepts and to appendix D for the complete grammar. Our current rules can be found in `sfl/behavior.sfl`. If a different ruleset shall be used this file will have to be edited.

### C.1 General Concept

As mentioned in chapter 8 SFLS is based on the standard coach language Clang. In order to deal with our needs we had to extend the language in order to express what we needed to implement a whole team's behavior. Still we maintained a compatibility to the original language making it easy for the coach language to be incorporated into our rule basis. For information on the standard coach language refer to chapter 7.

### C.2 Syntax

#### message types

We have five different message types that may be used to write rules: `advice`, `define`, `info`, `meta`, `freeform`. For our client we only used the first two as the others are not so crucial for a working client. Instead they rather mirror the types that Clang defines and are left for compatibility reasons.

The core type of message is `advice` since it describes rules in terms of conditions and directives which make up the client's behavior. Generally, each rule looks like this:

```
(advice
  (TIME-TO-LIVE
    (CONDITION)
    (DIRECTIVE1)
    (DIRECTIVE2)
    ...
    (DIRECTIVEn)
  )
)
```

The `TIME-TO-LIVE` sets the time as an integer number that this rule should be active measured in RoboCup frames starting at time 0. Use a figure  $> 6000$  to make sure that the rule will be valid during the whole game and eventual extra-times. A lesser value can be used to describe behavior that should only be used at the beginning of the game. Yet, actually this is just for compatibility reasons. The `time-condition` should be used to specify intervals that do not have to start at cycle 0.

## Conditions

In the `CONDITION` part a condition is expressed that is checked against the current world state. If the two match the first `DIRECTIVE` that refers to the agent will be executed. Below are a few examples of conditions:

- `true`  
a very simple condition that is of course always true
- `bowner our {5}`  
this rule matches if the player with the number 5 in our team has the ball
- `bowner opp {5}`  
this is true if the player with the number 5 in the opposing team has the ball
- `bowner our {0}`  
if it is not crucial that a certain player has the ball the number 0 is used. It matches if any member of our team holds the ball
- `ppos opp {0} 1 11 (arc (homepos) 15 30 0 360))`  
this condition describes a situation in which 1 and up to 11 opponents are in a ring (starting at 0 degrees and going around 360 degrees) that has an inner diameter of 15 and an outer diameter of 30 length units
- `ballInterceptable our {(closestPlayerToBall our) }`  
if the player on our team that is closest to the ball can intercept the ball this rule matches

Conditions may be negated by putting a `not` around the condition. Two or more conditions may be combined with an `and` or an `or`.

```
(and
  (CONDITION 1)
  (CONDITION 2)
  (CONDITION 3)
)
```

The equivalent is true for `or`.

## Directives

Each directive in our ruleset starts with a `do` followed by `our`<sup>1</sup>. Then at least one player number, situation-specific symbol, variable or 0 has to be given as well as a specific action.

---

<sup>1</sup>Directives with `dont` are not handled.

- `(do our {0} (pos (pt ball) 100) )`  
if the condition for this directive is true the whole team (represented by 0) is supposed to go to the ball position with power 100.
- `(do our { (closestPlayerToBall our) } (interceptball 100) )`  
this directive makes our closest player to the ball intercept the ball with power 100.
- `(do our {(closestPlayerToBall our)}(bto "their_goal" {s}) )`  
in this example the objective is to score a goal: our closest player to the ball shall shoot the ball into the region `their_goal` by method `score`.

The last example showed a use for the message type `define` which will be dealt with in the next section.

## Defines

Through defines it's possible to write down complex conditions, directives or regions once and through the given label refer to it easily elsewhere in the rule basis. Instead of writing the opponent's goal region each time a define is used:

```
(define (definer "their_goal_zone"
  (arc (pt 52.5 0) 0 18 0 360)
)
)

(define (definer "their_goal"
  (quad (pt 52.5 15) (pt 52 15) (pt 52 -15) (pt 52.5 -15))
)
)
```

Now each time the region around the opponent's goal or the goal itself is needed `their_goal_zone` or `their_goal` may be written as seen in the directive :

```
(do our {(closestPlayerToBall our)}(bto "their_goal" {s}) )
```

Likewise, conditions and directives may be defined and referred to. This way it is possible to write down complex situations like being in the offense or standard situations like a corner kick once and refer to them easily.

## C.3 Writing rules

When writing rules some things have to be kept in mind

- **priority rating**  
as can be read in chapter 8 the client chooses the current rule by a priority number. The higher the number the more important the rule or the more specific the rule. So each rule has to be headed by a number written in '<' '>'. Please refer to the file `sfl/behavior.sfl` from our released code for examples.

- **use of defines**

it has already been pointed out that the use of defines is rather powerful and its use is highly encouraged. It helps to make the rules more readable and keeps the danger of errors down.

- **not all is implemented**

although the grammar appears rather complete some features have not yet been implemented. This is documented in appendix D.

# Appendix D

## SFL - grammar

by Timo Steffens

The grammar of SFL is based on the standard coach language (Clang) [6]. Differences to Clang are commented.

```
<MESSAGE> -> <INFO_MESS>
            | <ADVICE_MESS>
            | <META_MESS>
            | <DEFINE_MESS>
            | <FREEFORM_MESS>

#Advice and Info messages
<INFO_MESS> -> (info <TOKEN_LIST>)
<ADVICE_MESS> -> (advice <TOKEN_LIST>)
<TOKEN_LIST> -> <TOKEN_LIST> <TOKEN> | <TOKEN>
<TOKEN> -> (<TIME> <CONDITION> <DIRECTIVE_LIST>)
            | (clear)
<CONDITION> -> (true)
            | (false)
            | (ppos NUM NUM TEAM UNUM_SET REGION)
            | (bpos <REGION>)
            | (bowner <TEAM> <UNUM_SET>)
            | (playm <PLAY_MODE>) |
            | (and <CONDITION_LIST>) |
            | (or <CONDITION_LIST>) |
            | (not <CONDITION>)
            | (action TEAM UNUM_SET ACTION) # Some player in UNUM_SET executes
                                           # ACTION. Not implemented in SFLS.
            | (time VALUE)                 # Servertime is VALUE (so may be
                                           # a variable)
            | (goal_diff VALUE)            # Goal difference is VALUE
            | (stamina TEAM UNUM_SET LEVEL) # Someone in UNUM_SET has the
                                           # specified stamina level
            | (eq VALUE VALUE)             # Used to compare variables and/or
                                           # constants
```

```

| (equinum UNUM UNUM)           # Used to compare uniform number
                                # variables
| (lt VALUE VALUE)             # less than
| (gt VALUE VALUE)             # greater than
| (state "STRING" "STRING")    # value of the state is the second
                                # STRING used to maintain states
                                # e.g. (state "ballStopped" "true")
                                # 2do maybe second STRING should be
                                # replaced by VALUE?
| (ballvelocity VARIABLE)      # ball has the specified velocity
| (ballinterceptable TEAM UNUM_SET) # Some player in UNUM_SET can
                                # intercept the ball
| (ballcatchable TEAM UNUM)    # intended for goalie, not
                                # implemented in SFLS
| "STRING"
<CONDITION_LIST> -> <CONDITION_LIST> <CONDITION>
<DIRECTIVE_LIST> -> <DIRECTIVE_LIST> <DIRECTIVE> | <DIRECTIVE>
<DIRECTIVE> -> (do <TEAM> <UNUM_SET> <ACTION>) |
| (dont <TEAM> <UNUM_SET> <ACTION>)
| (force <TEAM> <UNUM_SET> <ACTION>) # execute this rules as soon as
                                # encountered, ignore all other
                                # active rules
| "STRING"

<ACTION> -> (pos <REGION> [real]) | # Dash_power
| (pos <REGION>) |
| (home <REGION>) |
| (bto <REGION> <BMOVE_SET>) |
| (bto <UNUM_SET> PASS_MODE_LIST ) | # PASS_MODE
| (mark <UNUM>) |
| (markl <UNUM>) |
| (markl <REGION>) |
| (markl <REGION> <UNUM>) | # position agent between opponent UNUM
                                # and REGION
| (oline <REGION>) |
| (htype <HET_TYPE>)
| (state "STRING" "STRING") # saves states
                                # e.g. (state "ballStopped" "true")
| (interceptball [real]) # intercept ball with speed [real].
| (catchball) # intended for goalie, not implemented
                                # in SFLS
| "STRING"

<VALUE> -> integer | real | 'A'-'Z'
<LEVEL> -> low | mid | high # Levels for Stamina and such

<PASS_MODE_LIST> -> <PASS_MODE_LIST> <PASS_MODE> | <PASS_MODE>
<PASS_MODE> -> safe | risc | short | long # similar to BMOVE_TOKEN.

```

```

<PLAY_MODE> -> bko | time_over | play_on
              | ko_our | ko_opp | ki_out | ki_opp | fk_our | fk_opp
              | ck_our | ck_opp | gk_our | gk_opp | gc_our | gc_opp
              | ag_our | ag_opp
<TIME> -> [int]
<HET_TYPE> -> [int]
<TEAM> -> our | opp
        | both # not implemented in SFLS
        | teamOfFastestPlayerToBall # situation-specific symbol
        | teamOfClosestPlayerToBall #situation-specific symbol
        | (teamOfFastestPlayerToPlayer TEAM UNUM) #situation-specific symbol
        | (teamOfClosestPlayerToPlayer TEAM UNUM) #situation-specific symbol
<UNUM> -> [int(0-11)]
        | 'A'-'Z' # UNUMVARIABLE
        | (FastestPlayerToBall TEAM) #situation-specific symbol
        | (ClosestPlayerToBall TEAM) #situation-specific symbol
        | (FastestPlayerToPlayer TEAM TEAM UNUM) #situation-specific symbol,
          #first TEAM denotes the team of the resulting player,
          # second TEAM and UNUM describe target-player
        | (ClosestPlayerToPlayer TEAM TEAM UNUM) #situation-specific symbol
        | (BestPassPartner TEAM UNUM) #situation-specific symbol
          # best passpartner of UNUM in TEAM
          # TEAM and UNUM are ignored in SFLS
        | (BestDeckPartner TEAM UNUM) #situation-specific symbol
          # TEAM and UNUM are ignored in SFLS

<UNUM_SET> -> { <UNUM_LIST> }
<UNUM_LIST> -> <UNUM_LIST> <UNUM> | e

<BMOVE_SET> -> { <BMOVE_LIST> }
<BMOVE_LIST> -> <BMOVE_LIST> <BMOVE_TOKEN> | <BMOVE_TOKEN>
<BMOVE_TOKEN> -> p | d | c | s

<REGION> -> <POINT> |
          | (null)
          | (homepos) # homeposition of the evaluating(!) agent
          | (quad <POINT> <POINT> <POINT> <POINT>) |
          | (arc <POINT> [real] [real] [real] [real]) |
          | (reg <REGION_LIST>)
          | "STRING"
<REGION_LIST> -> <REGION_LIST> <REGION> | <REGION>
<POINT> -> (pt [real] [real])
          | (pt [real] [real] <POINT>)
          | (pt ball)
          | (pt <TEAM> <UNUM>)
          | (mult <POINT> <POINT> # multiply coordinates, used for
            # simple coordinate-arithmetic

```

```

| (plus <POINT> <POINT> # similar to point-relative,
# adds coordinates

<META_MESS> -> (meta <META_TOKEN_LIST>)
<META_TOKEN_LIST> -> <META_TOKEN_LIST> <META_TOKEN> | <META_TOKEN>
<META_TOKEN> -> (ver [int])

<DEFINE_MESS> -> (define <DEFINE_TOKEN_LIST>)
<DEFINE_TOKEN_LIST> -> <DEFINE_TOKEN_LIST> <DEFINE_TOKEN>
| <DEFINE_TOKEN>
<DEFINE_TOKEN> -> <CONDITION_DEFINE>
| <DIRECTIVE_DEFINE>
| <REGION_DEFINE>
| <ACTION_DEFINE>
| <PLAN_DEFINE> # used to collect several rules into one
# named tactic. untested in SFLS

<CONDITION_DEFINE> -> (definec "[string]" <CONDITION>)
<DIRECTIVE_DEFINE> -> (defined "[string]" <DIRECTIVE>)
<REGION_DEFINE> -> (definer "[string]" <REGION>)
<ACTION_DEFINE> -> (definea "[string]" <ACTION>)
<PLAN_DEFINE> -> (defineplan "[string]" <TOKEN_LIST>) # see above

<FREEFORM_MESS> -> (freeform "[string]")

```

## Appendix E

### netif.C

```
deque<std::string> orca_udp_buffer(10); int init_udp = 0;

int wait_message(char *buf, Socket *sock) {
    if (receive_message(buf, sock) == 1) {

        if(strncmp(buf,"(init",4) == 0){
            init_udp=1;
            return 1;
        }
        while(receive_message(buf, sock) == 1){
            if(strncmp(buf,"(init",4) == 0){
                init_udp=1;
                return 1;
            }else{
                orca_udp_buffer.push_back(buf);
            }
        }
    }
}

else for (int i = 0; i < 100; i++) {

    if (receive_message(buf, sock) == 1){

        if(strncmp(buf,"(init",4) == 0){
            init_udp=1;
            return 1;
        }
        while(receive_message(buf, sock) == 1){
            if(strncmp(buf,"(init",4) == 0){
                init_udp=1;
                return 1;
            }else{
                orca_udp_buffer.push_back(buf);
            }
        }
    }
}
```

```

    }
    }
    my_error("sleeping, waiting for message");
    usleep(50000);
}
return 0;
}

(...)

int receive_message(char *buf, Socket *sock) {
    int          n, servlen ;
    struct sockaddr_in  serv_addr ;
    if(!(orca_udp_buffer.empty()==0) && init_udp == 1){
        // fprintf(stderr, "hole was vom puffer\n");
        buf = (char *) (orca_udp_buffer.front()).c_str();
        orca_udp_buffer.pop_front();
        return 1;
    }

    servlen = sizeof(serv_addr) ;
    n = recvfrom(sock->socketfd, buf, MAXMSG, 0,
        (struct sockaddr *)&serv_addr, (socklen_t *)&servlen);
}

(...)
```

# Appendix F

## Authors

<b>Section</b>	<b>Author</b>
1	Andreas G. Nie
2	Philipp Hügelmeier
3	Andres Pegam and Marco Diedrich
4.1 and 4.2	Sean Buttinger
4.3	Angelika Hönemann
4.4	Leonhard Hennig and Philipp Hügelmeier
4.5	Timo Steffens
5	Andres Pegam
6	Collin Rogowski
7	Timo Steffens
8	Timo Steffens
8.3.1	Sean Buttinger
9.1	Collin Rogowski
9.2 and 9.3	Leonhard Hennig
10	Andres Pegam
11	Timo Steffens and Andreas G. Nie
A, B and C	Andreas G. Nie
D	Timo Steffens
E	Philipp Hügelmeier

# Bibliography

- [1] Homepage of CVS: <http://www.cvshome.org>.
- [2] Homepage of FC Portugal 2000: <http://www.ieeta.pt/robocup>.
- [3] Justin A. Boyan and Andrew W. Moore. Generalization in reinforcement learning: Safely approximating the value function. In G. Tesauro, D. S. Touretzky, and T. K. Leen, editors, *Advances in Neural Information Processing Systems 7*, pages 369–376, Cambridge, MA, 1995. The MIT Press.
- [4] Justin A. Boyan and Andrew W. Moore. Learning evaluation functions for large acyclic domains. In L. Saitta, editor, *Proceedings of the 13th International Conference on Machine Learning*, Bari, Italy, 1996. Morgan Kaufmann.
- [5] Sean Buttinger, Marco Diedrich, Leonhard Hennig, Angelika Hoenemann, Philipp Huegelmeyer, Andreas Nie, Andres Pegam, Collin Rogowski, Claus Rollinger, Timo Steffens, and Wilfried Teiken. The Dirty Dozen Team and Coach Description. In Andreas Birk, Silvia Coradeschi, and Satoshi Tadokoro, editors, *RoboCup-2001: Robot Soccer World Cup V*. Springer, Berlin, 2002. (to appear).
- [6] Mao Chen, Ehsan Foroughi, Fredrik Heintz, ZhanXiang Huang, Spiros Kapetanakis, Kostas Kostiadis, Johan Kummeneje, Itsuki Noda, Oliver Obst, Patrick Riley, Timo Steffens, Yi Wang, and Xiang Yin. Soccerserver manual v7, 2001.
- [7] Thomas G. Dietterich. Machine-learning research: Four current directions. *The AI Magazine*, 18(4):97–136, 1998.
- [8] Omid Aladini et al. Hella respina team and coach description. In Andreas Birk, Silvia Coradeschi, and Satoshi Tadokoro, editors, *RoboCup-2001: Robot Soccer World Cup V*. Springer, Berlin, 2002. (to appear).
- [9] Yang Yang et al. Wright eagle team and coach description. In Andreas Birk, Silvia Coradeschi, and Satoshi Tadokoro, editors, *RoboCup-2001: Robot Soccer World Cup V*. Springer, Berlin, 2002. (to appear).
- [10] A. Intelligence and P. Cohen. Empirical methods for artificial intelligence, 1995.
- [11] Vipin Kumar. Algorithms for constraint-satisfaction problems: A survey. *A.I. Magazine*, 13(1):32–44, Spring 1992.
- [12] Scerri P. and Ydren J. End user specification of robocup teams. In Veloso, Pagello, and Kitano, editors, *RoboCup-99: Robot Soccer World Cup III*. Springer, Berlin, 2000.

- [13] Luis Paulo Reis and Nuno Lau. FC Portugal Team Description: Robocup 2000 Simulation League Champion. In Tucker Balch Peter Stone and Gerhard Kraetzschmar, editors, *RoboCup-2000: Robot Soccer World Cup IV*. Springer, Berlin, 2001.
- [14] Luis Paulo Reis and Nuno Lau. Coach unilang - a standard language for coaching a (robo) soccer team. In Andreas Birk, Silvia Coradeschi, and Satoshi Tadokoro, editors, *RoboCup-2001: Robot Soccer World Cup V*. Springer, Berlin, 2002. (to appear).
- [15] M. Riedmiller, A. Merke, D. Meier, A. Hoffmann, A. Sinner, O. Thate, and R. Ehrmann. Karlsruhe brainstormers – a reinforcement learning approach to robotic soccer.
- [16] Patrick Riley. Classifying adversarial behaviors in a dynamic inaccessible multi-agent environment, 1999. CMU-CS-99-175.
- [17] Patrick Riley, Paul Carpenter, Gal Kaminka, Manuela Veloso, Ignacio Thayer, Robert Wang, and Patrick Markiewicz. ChaMeleons and OWL simulator team and coach. In Andreas Birk, Silvia Coradeschi, and Satoshi Tadokoro, editors, *RoboCup-2001: Robot Soccer World Cup V*. Springer, Berlin, 2002. (to appear).
- [18] Patrick Riley, Manuela Veloso, and Gal Kaminka. Towards any-team coaching in adversarial domains. 2002. submitted to AAMAS 2002.
- [19] Peter Stone, Patrick Riley, and Manuela Veloso. The CMUnited-99 champion simulator team. In Veloso, Pagello, and Kitano, editors, *RoboCup-99: Robot Soccer World Cup III*, pages 35–48. Springer, Berlin, 2000.
- [20] R. Sutton and A. Barto. Reinforcement learning, 1998.
- [21] G. Tesauro. A self-teaching backgammon program, 1994.
- [22] John N. Tsitsiklis and Benjamin Van Roy. An analysis of temporal-difference learning with function approximation. Technical Report LIDS-P-2322, 1996.