Creating agents for the RoboCup Soccer Simulator using evolutionary techniques

Phil Bradfield

BSc Cognitive Science (Industry)
Session 2006/2007

The candidate confirms that the work submitted is their own and the appropriate credit has been given where reference has been made to the work of others.

I understand that failure to attribute material which is obtained from another source may be considered as plagiarism.

(Signature of student)_____

Summary

The RoboCup project aims to encourage progress in the fields of artificial intelligence and robotics by providing a standard problem for developers to solve. This problem is a game of football (soccer), which can be played either for real by physical robots, or in simulation by software agents. The aim of this project was to use evolutionary computation methods, specifically genetic programming, to develop agents to operate within the framework of the RoboCup Soccer Simulator.

Before any agents could be created, however, an appropriate software architecture had to be developed. Over the course of the project, it became clear that the implementation of this architecture was going to be much more complex than was initially thought. As a result, the aim of the project was reduced to that of producing a software framework for the evolutionary creation of RoboCup agents. Even this, however, proved impossible in the timeframe of the project.

This report details the process of creating the framework, and examines some of the issues involved in using evolutionary computing techniques in the RoboCup Soccer domain. It also details the final state of the framework and the work which would be necessary to bring it to a conclusion, comments on the project process and management and gives some thoughts on how this area of research could be taken forward in the future.

Acknowledgements

I owe a large debt of gratitude to all of the following people, without whom this project would never have got as far as it did:

- My supervisor Dr. Andy Bulpitt, my assessor Dr. Marc de Kamps and my tutor Prof. Roger Boyle for their academic support and their helpful input and advice;
- All the helpful people who answered my questions (even the stupid ones) on the local.lang.c-plus-plus newsgroup;
- Matt for agreeing to help proof-read the write up when he had far more interesting things to do;
- My family, friends and housemates for their personal support over the course of the year;
- My fellow coggies,who provided some much needed laughs during the many hours spent in the ENIAC lab working on the project;
- And most of all my girlfriend Deborah, who is largely responsible for preventing me from going completely insane during the course of the project.

Contents

Chapter 1

Introduction

1.1 Project Aims and Deliverables

The overall aim of this project was to use techniques from evolutionary computation (see section 2.2) to produce agents for the RoboCup soccer simulator (see section 2.1). In order to reach this goal, the problem was divided into several sub-problems, each of which was intended to be delivered as a product:

1. Create a framework which would facilitate the use of evolutionary computing in developing RoboCup agents;

2. Use the above framework to create a simulated agent capable of performing a simple individual behaviour, specifically being able to dribble the ball to a specific point;

3. Use the framework to create a group of agents capable of performing a more complex task involving co-operative or antagonistic interaction between agents; specifically, performing well at keepaway soccer (see section 2.1.5.

It became clear very early on in the project that the grand aim of having highly-developed agents performing complex, interactive behaviours was not going to be achievable in the short timeframe available for the project. There was one simple reason for this: the amount of work involved in simply

creating the framework (aim no. 1) was much larger than anticipated. It appeared from my research that much of the work had already been done, and that it would not require a lot of effort to link the disparate elements of the framework together. This impression was completely false: creating the framework was far from simple. Chapters 4 and 5 go into detail about the design and implementation of the framework.

Because of this, the aims of the project were reduced to simply completing the framework and, if possible, trying to evolve some simple agents. This meant that some of the preparation for the project became unusable, as it had been focussed on the keepaway task (aim no. 3); aim no. 2 was intended merely as a staging post between aims 1 and 3, allowing me to experiment with the framework using a relatively simple problem before moving on to tackle the more complex issue.

In time however, even the reduced aim of simply completing the framework proved unobtainable. I believe that I came close to having a functioning framework which would have allowed me to start evolving agents, but I simply ran out of time. The final state of the product is evaluated in chapter 6.

1.2 Project Schedule

This is the project schedule which was agreed upon as part of the Mid-Project Report (submitted in December 2006).

January 15th 2007 Final Semester One exam. Little FYP progress expected before this point.

January 26th 2007 Complete evaluation of GP toolkits; complete familiarisation with selected toolkit.

February 2nd 2007 Complete work on toolkit-team interface.

February 23rd 2007 Complete implementation of agents' individual behaviours.

March 16th 2007 Deadline for progress meeting.

April 9th 2007 Complete implementation of interactive agents.

April 24th 2007 Complete write up.

April 25th 2007 Deadline for project hand-in.

1.3 Report Structure

This report consists of 7 chapters and 4 appendices:

Chapter 1 introduces the problem and the aims of the project.

Chapter 2 gives background information about the issues the project aimed to address. This includes salient information about the RoboCup soccer server and the agents which use it; about evolutionary computation and especially genetic programming; and about previous research which has used evolutionary computation to develop RoboCup agents.

Chapter 3 details the preparatory steps taken before implementation could be designed, including the selection of off-the-shelf code for use in the project and the specification of problem parameters.

Chapter 4 details the design of the project framework and the plan which was drawn up for the management of the implementation.

Chapter 5 describes the process of implementing the project, the obstacles encountered and how they were overcome, and how the off-the-shelf products were first modified and then linked together.

Chapter 6 describes the final state of the project (as at the time of writing), and evaluates the framework produced and how close it is to completion. Other aspects of the project are also evaluated, including research conducted for the project and the way that the project was managed.

Chapter 7 gives some thoughts on how this research could be taken further in the future.

Appendix A reflects on the project exerience from a more personal angle, including the mistakes I made during the project and lessons I have learnt from them.

Appendix B provides a list of the skills which come pre-implemented with the TriLearn RoboCup team (see section 3.1.1.2), which may be a useful reference during the reading of the report.

Appendix C provides a brief description of some of the custom data types created for TriLearn and used in the implementation.

Appendix D contains the BEAGLE configuration file which would have been used to control the evolution process.

Chapter 2

Research

2.1 The RoboCup Project

The Robot World Cup Initiative (RoboCup) began in 1993. It aims to encourage research in robotics and artificial intelligence in two ways: firstly, by setting a standard problem for researchers to work on, and secondly by raising the profile of these areas by setting a goal which is broadly appealing to the general public. This goal is to make the following scenario come true:

> "By mid-21st century, a team of fully autonomous humanoid robot soccer players shall win a soccer game, complying with the official rules of FIFA, against the winner of the most recent World Cup." [8]

The RoboCup project currently has three domains: RoboCup Soccer, RoboCup Rescue and RoboCup Junior, of which only RoboCup Soccer is relevant to this project. There is an annual world championship of RoboCup, in which teams in all three domains compete against each other, and researchers can show off their latest innovations.

2.1.1 The RoboCup Soccer Domain

The RoboCup Soccer domain contains several different disciplines. At the time of writing, these are:

- 2D Simulation League
- 3D Simulation League
- Coach Simulation League
- Small Size Robot League
- Medium Size Robot League
- Four Legged Robot League
- Humanoid Robot League

The 2D and 3D Simulation Leagues both involve teams of 11 simulated autonomous robots (agents) playing soccer against each other. Generally speaking, the teams in these simulation leagues show much more realistic and advanced teamwork and cooperative behaviour, as they are not hampered by the limitations of actual robotic bodies. As such, the 2D and 3D Simulation Leagues make an excellent test bed for trying out new methods in multi-agent behaviour. In order to keep the problem as simple as possible, this project shall use the 2D, rather than the 3D Simulation. The Coach Simulation is different; the Coach League represents an attempt to create an omnipotent 'coach' which can give instructions to the players in real time, in order to try to help the team's play. This coach may come in useful later in the project, as coaches have access to all the information coming out of the soccer server, and as such may be useful in evaluating my own evolved teams. See section 2.1.4.

2.1.2 The RoboCup Soccer Server

Teams entered into the RoboCup 2D Simulation League use the RoboCup Soccer Server (`rcsoccersim`, [3]). This is a software system which simulates the playing field on which the agents play. Under the soccer server system, each of the 22 agents involved in a game runs as a seperate process, and each individual process communicates with the server via a Unix socket. Every 100 milliseconds, each agent tells the soccer server which actions they wish to carry out; every 500 milliseconds the server supplies the agents with perceptual information. This means that agents are often operating with information which is slightly out-of-date. All the perceptual information which the soccer server sends to the agents has random noise added to it, so the information is also not completely reliable. To top this off, the number of actions that a single agent can take in a single 100ms timestep is strictly limited; for example, an agent can only `kick`, `dash` or `turn` once per timestep (see section 2.1.3 for more information on agent commands).

This all adds up to RoboCup being a dynamic, complex, multi-agent environment containing both antagonistic and cooperative elements, in which agents have to operate effectively despite their perception being neither complete nor reliable. This represents a very difficult challenge for AI developers, and a good vehicle for testing new techniques in AI.

2.1.3 The RoboCup Agent

Each agent has its own range of perception, and can perceive information via visual, aural and body sensors. The body sensor is the agent's equivalent of proprioception, which is necessary because the agent's simulated "head" can be turned independently of its body, allowing the agent's body to be facing in one direction while its head is turned so that it is looking in a different direction. The body sensor allows the agent to sense this. The aural sensor allows for direct communication between agents, akin to human players shouting to each other on the pitch. The amount of information which can be transmitted over the auditory channel is deliberately limited in order to prevent agents becoming aware of more perceptual information than an equivalently situated human player would; for instance, one player could not send out all the information that it can perceive about the world to all other players, as this would violate the autonomy of the agents. The visual sensor is the most important of the three, just as it is to a human football player. It is through this sensor that the agent acquires information about the objects on the field: positions of the goals, the ball, other players, the edges of the pitch etc, and also the velocity of moving objects.

The agent's visual perception is limited by its field of vision (typically a 90 degree cone, although because this is a parameter of the soccer server it can be altered), and so the agents have to rely on incomplete information (ie they won't know the positions of objects outside their field of vision). It is possible that agents may be able to make an informed guess about an object's current position and velocity based on the object's last known position and velocity, but as this can be classified as part of the agent's "intelligence", it is of course up to the developer to implement it.

In addition to receiving sensory information from the server, an agent must also give the server action commands. The actions which an agent can take are: `kick`, `dash`, `turn`, `catch` (goalkeepers only), `say` and `turnNeck`. At each timestep, the agent can execute a maximum of three actions: one say action, one turnNeck action and one kick, turn, dash or catch action. None of these are compulsory. An agent also has an internal stamina model. Stamina decreases every time that a player dashes and is very gradually replenished whenever a player takes some other action. A player with high stamina can

dash faster than a player with low stamina.

2.1.4 The RoboCup Coach

There are two types of coach in RoboCup simulations: an online coach and a trainer. Roughly speaking, the online coach is equivalent to a human manager trying to instruct his team from the touchline during a match, and the trainer is like a coach supervising a training session. Both coaches have full access to visual information about every object in the soccer server, without noise added; the trainer also has full access to all auditory information, again without noise added.

The online coach is only used during matches, and gives instructions to the players concerning strategy. The online coach's communication with the players is done via the server's auditory model, and is limited to sending at most four messages per 300 cycles, ensuring that developers cannot create an online coach which centrally controls all the team's players. The online coach can also make a limited number of substitutions at any point that it wishes.

The trainer is the more interesting of the two coaches from the point of view of this project. The trainer has far more power than the online coach. As well as being able to send an unlimited number of messages via the auditory channel and make unlimited substitutions, the trainer can at any time:

- move ("teleport") players or the ball from one point on the pitch to another;
- change the play mode (ie start and stop play, award a free kick to either side, etc);
- reset all players' stamina, recovery and effort to their original values.

The trainer is intended to allow the automation of machine learning processes. For instance, in the case of this project I would be able to tell the trainer the conditions under which the simulation should be started and stopped, and the trainer would be able to completely control the scenario. In addition, the trainer's access to noiseless data from the server means that whatever data I try to use for my fitness function would be readily available to it, and it would be able to send the data back to the process controlling the evolution in order for that process to calculate the agent's fitness.

2.1.5 Keepaway Soccer

Keepaway soccer is a subproblem of the larger RoboCup soccer problem, and provides a much simpler (though not simplistic) proposition than the full soccer scenario. It was first proposed by Stone and Sutton as a testbed, allowing researchers to directly compare the efficacy of one machine learning approach

to that of others [24]; however it would also serve my purpose well as a reduced version of the RoboCup problem.

In keepaway soccer, one team of players (known as the *keepers*) tries to keep possession of the ball for as long as possible, while another team (the *takers*) try to take it off them. The performance of the teams is measured by how long it is from the start of the simulation until one of the takers touches the ball; the keepers attempt to maximise this time, the takers to minimise it. The exercise takes place within a restricted area; if the ball leaves this area then the keepers lose and the clock stops.

The main parameters for the keepaway scenario are the number of keepers, the number of takers and the size of the area they are playing in. One version of keepaway used by several research groups has three keepers and two takers playing in an area of size 20 metres x 20 metres ("3v2 keepaway on a 20x20 pitch"). This smaller number of parameters and objects obviously helps to simplify the problem and make it solvable in a shorter space of time than the full soccer problem. Another point in keepaway's favour is that it makes the derivation of a fitness function for the evolution very easy, as the performance of the agents is simply measured by the length of time that the keepers can keep the ball; see section 3.3.2.

Keepaway has already been used by researchers looking at machine learning techniques, including relational reinforcement learning [33], temporal difference reinforcement learning [25] and behaviour transfer [28]. Interestingly, evolutionary algorithms have also been applied to keepaway, with very successful results. Di Pietro et al [18] used evolutionary algorithms to create agents based on a representation of the keepaway state-space defined in a paper by Stone, Sutton and Singh [26]. Their approach differed significantly from my proposed approach (based on genetic programming; see section 2.2), but their results did show the potential of evolutionary learning processes: their best keeper agents managed to keep the ball for an average of 24.8 seconds before the takers took it off them, representing an improvement of 12.0 seconds over the initial population. By contrast, Stone et al's best keepers (developed using reinforcement learning, but using an identical state-space representation) achieved an average retention time of 14.5 seconds, an improvement of 9.0 seconds over their original agents. These results show that evolutionary computation can be an extremely effective method of generating keepaway agents, and so I am optimistic that an approach using genetic programming can also come up with a good solution.

2.2 Evolutionary Computing

Evolutionary computing (EC) is an umbrella term for several different computing techniques which take their inspiration from Darwinian evolution. Roughly speaking, EC can be broken down into four categories [7]:

- Genetic Algorithms (GAs)
- Evolution Strategies (ES)
- Evolutionary Programming (EP)
- Genetic Programming (GP)

Of these four, the one which is best suited to use in the RoboCup Soccer domain is GP. GP is a trial-and-improvement style of programming; it involves generating programs (usually randomly), running each one and measuring its performance, and then generating new programs based on the old ones, with the newly generated programs more likely to be based on the higher performing programs than on the lower performing ones.

This type of process is standard in evolutionary computing; however GP is unique in that it generates entire programs. The other three variants listed above can only find the most appropriate parametric values for programs which have already been written; this type of ability is not really very helpful in the context of programming agents for RoboCup, as it would require the structure of the agent programs to be determined by the programmer. Trying to determine the best structure out of the practically infinite alternatives would be practically impossible for a human developer, and as such we need an automated way of trying to find it. This points to GP as being the most promising possibility, as it is the only one of the four which allows us to automatically evolve both the program's values and its structure.

2.2.1 Genetic Programming

Genetic Programming was first described by Nichael Cramer [5] and popularised several years later by John Koza [12]; Figure 2.1 shows a high-level view of the procedure involved in GP. GP programs are expressed as syntax trees [14], rather than as standard code. Each node in the tree represents a *primitive*, which can be either a *terminal*, which takes no input, or a *function*, which takes at least one input. A terminal represents either an independent variable, a function with arity 0 or a constant; constants are usually set randomly. The functions specify operations to perform on the terminals, and the results of

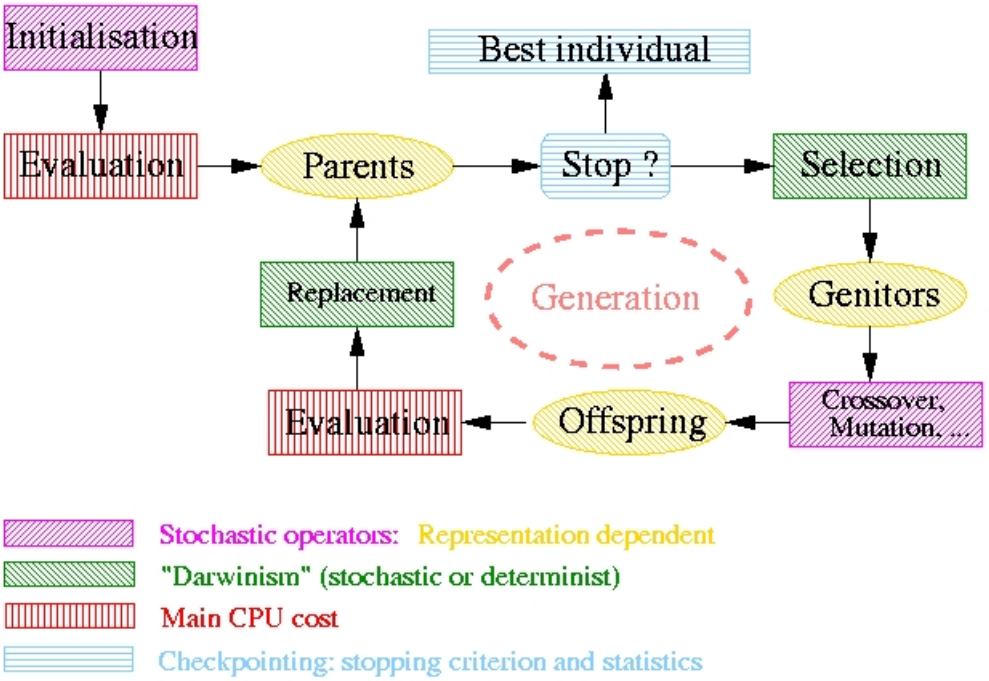

Figure 2.1: Flowchart of the genetic programming procedure. From [32]

the functions are propogated up through the tree to the root node, which is where the program's final result emerges.

At the beginning of a run, a number of trees (*individuals*) are randomly generated from a specified set of functions and terminals (*initialisation*). These form the initial population. These trees are then run, and the output they give is analysed. Each tree is then given a score (*fitness*), depending on how close its output is to that which is desired. The fitness score is determined by a human-defined *fitness function*. Defining the fitness function is often one of the hardest parts of using GP, especially for complex problems such as this: although a human may be able to watch a match and say "Team A performed better than Team B", it can be very difficult to translate this into terms which a computer can understand. More is said about fitness functions in section 2.3.

Once the fitness of the initial population has been measured, the individuals in the population are ranked by their fitness. The program then attempts to create the next *generation* of individuals by using the standard EC operations of *crossover* and *mutation*.

Crossover in GP involves selecting two individuals from the population, picking a random node in

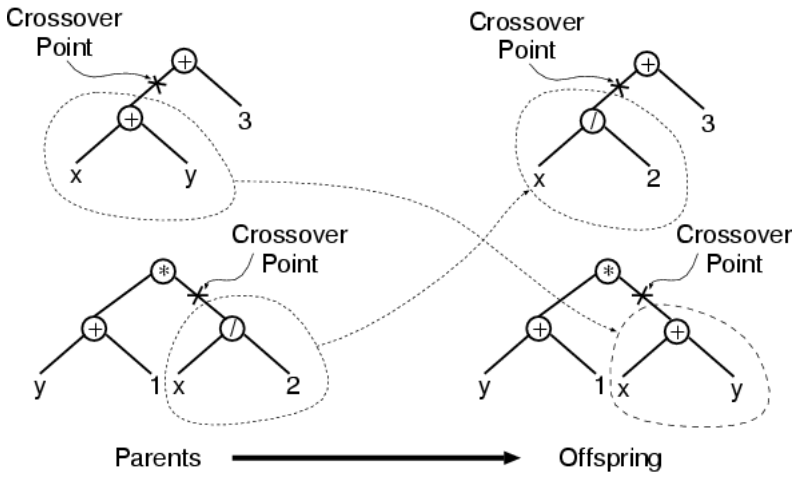

Figure 2.2: Diagrammatic representation of crossover in GP; from [14]

each tree and swapping the subtrees below those nodes. Thus, the subtree from individual 1 is now part of individual 2, and vice versa (see Figure 2.2). The new trees formed by the crossover go on to be part of the next generation. GP mutation involves picking a single individual from the old population, picking a random node in that tree, and then replacing the subtree below that node with a randomly generated one. This new tree moves into the new generation. These methods, if simply used as stated above, can lead to trees becoming very large, very quickly (this is known as *bloat*). This reduces the efficiency of both the algorithm and the eventual solution. To combat bloat, it is common to specify a maximum branch depth for all the trees in the run; any crossover or mutation operation which would increase the depth to more than this limit is blocked from occurring.

The likelihood of any one of the trees from the old population being selected for either of these operations depends entirely on its fitness score; the fitter an individual is, the more chance it has of being selected.

The process of crossover is intended to be analogous to sexual reproduction, while mutation is analogous to asexual reproduction. The idea that the individuals which are fittest (ie most well suited to the task at hand) are most likely to survive and breed and thus create the next generation is directly inspired by Darwinian evolution. In EC, this probability of survival is represented by assigning each individual a probability of selection (usually based either on its fitness score or its fitness ranking), and then randomly picking an individual based on those probabilities. The actual algorithm which assigns the probabilities to the individuals can vary; it is a matter of choice for the programmer, based on the

specific problem at hand.

This process of testing the individuals; performing selection, crossover and mutation; creating the next generation; and running and testing this new generation is repeated until some stopping condition is reached. This stopping condition will generally be either that the population has become sufficiently good at its task (ie the fitness either of an individual or of the generation as a whole has reached some predefined threshold) or that the evolution has completed a specified number of generations. If the evolutionary process has worked, then the individuals in the final generation should be much more proficient at the task than the individuals in the first generation were, and hopefully proficient enough to be usable in applications.

There is one major problem with regards to applying GP to RoboCup. In standard GP, it is assumed that any non-terminal will be able to take as input the output from any terminal or function, effectively meaning that it is only possible to have one data-type in a program. For some problems (eg mathematical functions) this is acceptable; however in the case of RoboCup it will clearly not be. The diversity of information which the agent will be dealing with is very large; it could include the position of an object (represented either by an (x,y) co-ordinate or a vector); a floating point number, such as a speed; an object representing a skill to execute; an object representing another player etc. If this information were to be put through a standard GP process, then no regard would be paid to types when the GP trees were built. Thus, you could end up having a floating point number being given as an argument to a function which is expecting a skill command, resulting in a meaningless tree which would most likely either crash horribly or produce very strange output. The solution to this problem is strongly typed genetic programming.

Strongly Typed Genetic Programming

Strongly typed genetic programming (STGP) [17] is an extension of GP which allows for primitives of different types to be used in the same tree. Essentially, every primitive is explicitly assigned a data-type which it will output, and function primitives additionally have expected-types assigned to each of the inputs which they take. Any time when a tree is created or changed, the STGP algorithm will take note of these assignations and make sure that the types match throughout. The parts of the GP process affected by this are initialisation, mutation and crossover, as these are the only times during the process when the trees are changed.

This ability to use multiple data-types allows the programmer's representation to be far closer to

real-life than would otherwise be the case (consider how realistic it could ever be to represent a football player as, say, a floating point number). It also helps to reduce the search space of the problem by restricting the allowed make up of the GP tree. These reasons mean that not only my work, but also the vast majority of the previous research using GP for RoboCup, has had to use STGP.

2.2.2 Toolkits for Genetic Programming

Many toolkits are available to make GP processes simpler to implement. They are available in a number of different languages, from Java to LISP to PUSH. As the RoboCup Soccer Server and most RoboCup agents are written in C++, it makes most sense to me to try to use a GP toolkit written in C++, as this is likely to make interfacing between the server, the agent and the toolkit much, much easier. The toolkits considered are described in section 3.1.2.

2.3 Previous Use of Evolutionary Computing Techniques In RoboCup Soccer

There have been several documented attempts to use genetic programming techniques to develop teams for RoboCup. Some of these resulted in teams which competed in RoboCup Championships, whilst others were simply research projects which intended to evaluate the effectiveness of evolving RoboCup teams under different conditions. All of these projects used genetic programming (GP) for this task. This section shall provide an overview of this literature.

2.3.1 Competitive Teams

Three genetically programmed teams have competed in RoboCup finals tounaments:

- U.Maryland in 1997, developed at the University of Maryland [16];
- Darwin United in 1998, developed at the University of California, Berkeley [1];
- Wroclaw2002 in 2002 developed at Wroclaw University of Technology [22].

Of these three, only U.Maryland made it past the first round; it is worth noting that this may be at least partly because the number of entrants in 1997 (the first RoboCup) was far smaller than in subsequent years.

As the most recent of the three, I had hoped that Wroclaw2002 would provide me with plenty of helpful information. However this was not the case; the only English-language documentation available relating to Wroclaw2002 is the team description submitted to the RoboCup organisers. This document does not go into much detail about the development or functioning of the team, except to say that the team was composed of ten homogeneous agents, and that most of their low-level behaviours were developed by evolution. Quite how far the development got is unknown, although it appears that it did not reach the stage of looking at much high-level, coopoerative behaviour; the developers' English website [23] (last updated in November 2003, nearly a year and a half after RoboCup2002) states that "Cooperative sequences and deliberative planning for small formations are still under construction".

On the other hand, U.Maryland and Darwin Utd are usefully documented. Both of the teams enjoyed moderate success in the Robocup competitions in which they competed, winning one or two games before being eliminated. Both projects started from a similar setup: they gave each agent a set of basic individual behaviours (*skills*), such as passing, shooting and intercepting, before trying to evolve cohesive team behaviour. In the case of Darwin Utd, these individual behaviours were hand-coded; U.Maryland had some hand-coded and some evolved. U.Maryland's original intention was to evolve all the agents' lower level behaviours, but they found that some skills, such as intercepting, were very difficult to evolve successfully, and that hand-coded methods produced better results.

This is where the similarities between the two ended though; the two teams were evolved very differently. U.Maryland used co-evolution, playing off generated teams against each other and allowing only the winner to reproduce. Darwin Utd's evolution was more complex: each of their generated teams would play against a series of three opponents, each of increasing ablity. The generated teams' first game was on an empty pitch, with no opposing players at all, and the generated team had to score within thirty seconds in order to be allowed to play against the next opponent: a team whose players did not move, but span on the spot and kicked the ball if it came near them. If the generated team succeeded in scoring within thirty seconds against this team, then their final match was against the RoboCup World Championship winners from the previous year (1997), a team called AT-Humboldt. The teams which were most successful in each generation (ie progressed the farthest through the series of opponents, and got the best results against them) were given a higher reproductive probability.

This approach has one big flaw, namely that it is too geared to evolving a team to play against one particular opponent in AT-Humboldt. It is highly possible that, using this type of evolution strategy, evolved teams would develop strategies which were highly successful against the opponent team used

in the evolution, but useless against other teams, making for an extremely fragile implementation. One way to avoid this would be to have a pool of opponent teams, of which one was picked at random to play against the evolved teams. This would avoid the problem of overspecialisation, although it may be introducing a bit too much randomness into the evolution, making the selection process more about luck (ie getting drawn against a poorer opponent team) than about the skill of the generated teams. Luke's co-evolution approach does not share these drawbacks, as the generated teams always faced different opponents during their evolution. The main possible problem with co-evolution is that the population could become too insular and lacking in diversity, again leading to a fragile final result. To avoid this, a reasonably high level of mutation is needed to make sure that the population does not converge on a sub-optimal solution early on (such as the kiddie soccer solution; see 2.3.2 below) and then fail to get out of it.

In a retrospective paper, Luke evaluated the strengths and weaknesses of his approach [15]. He concluded that his team could have been improved by having a larger population size; having more frequent competition between teams; evolving individual agents rather than full teams; having a more generalised base set of atomic functions from which to work; and providing the agents with some kind of internal state. It is possible that my work, despite Luke's warnings, will share some of these limitations. Those which are most likely to come into play are those of population size and of limited competition (both of which are restricted by the time and computational resources available), and of the lack of an internal state. Koza [13] recommends a population size of 4000 for a typical problem, a number which is simply unworkable in this scenario. An internal state is something that I feel would probably be highly beneficial to any RoboCup team; however, it may prove to be a complication too far in the time available to me.

2.3.2 Research Projects

In addition to these full teams, there have also been at least two research projects which have looked at using GP to create teams for RoboCup. Both of these projects took an experimental approach to the problem, comparing different setups to find out which produced the agents with the most favourable behaviour.

Wilson [34] compared the performance of different selection methods and different primitive sets when using GP for RoboCup. When he tried to evolve a team using a starting set of only very low level functions (ie the functions provided by the soccer server, such as `kick`, `turn` and `dash`), he found that

using tournament style selection produced very poor results. Most teams failed to even kick the ball, let alone score, and many of the teams which came out as the "best of generation" only used the `kick` function; they never actually moved at all. The selection method used in the experiment was to play teams against each other and eliminate the loser; if the score in any match was a draw, the team with the least successful passes was eliminated; if this was also equal, the team which kicked the ball the fewest times was eliminated. Wilson attributed the failure of this experiment to this selection method; it meant that teams were often selected for reproduction completely at random, as the majority of teams were not good enough to even kick the ball at all, resulting in their scoring zero on all fitness measures.

This does not mean that tournament selection may not be useful in other situations; once the teams have progressed far enough to reliably score (or at least kick the ball) in most games, then tournament selection would probably work perfectly well. The problem only arose because the basic teams were so poor at even the most fundamental aspects of playing football. Tournament selection is, in the context of soccer, a very obvious selection method to use, as it looks at the most important factor in any soccer match: which team scores the most goals. But Wilson's experiment shows that this is not necessarily applicable at all stages of the evolutionary process; until the population reaches a certain level of competence, some other measure of fitness is necessary, just as a young child playing football for the first time has his ability measured differently to an older, more experienced player. This implies that using a constant fitness function, which is the standard in any GP, may not be so useful in this scenario, and that a fitness function which adapts as the population evolves may well provide better results. More is said about this in chapter 7.

Wilson's next experiment looked to see whether the results could be improved by including in the initial population a small number of hand coded teams which already had a small measure of playing ability. The hand coded players simply ran towards the ball and kicked it towards the opponents' goal; the selection method used was Reynolds' new-versus-several method [19]. Wilson found that, because the hand coded teams were, despite their simplicity, much more advanced than the randomly generated teams they were competing against, they quickly took over the entire population, meaning the population's diversity was very quickly reduced to almost nothing. With hindsight, this would seem an inevitable consequence of not creating a level playing field for all teams at the start of the evolutionary process.

Finally, Wilson looked at using higher level functions in his representation, eg a `kickTo` function which told players to sequentially turn towards the ball, move towards the ball, and kick the ball. This

experiment resulted in what appears to be a common suboptimal solution to the RoboCup problem: the "kiddie soccer" solution. This solution involves all players following the simple strategy of chasing after the ball and, when they reach it, kicking it towards the opponent's goal. As well as Wilson, Luke mentions this result as being one that he often encountered in his own efforts at evolving RoboCup agents [16], implying that kiddie soccer can be hard to avoid. It was the frequency with which the kiddie soccer solution kept turning up that encouraged Luke to use a very high mutation rate in his experiments; this helped the evolution to escape from the local minima and find a better solution.

The second research project was conducted by Aronsson [2]. He tested both heterogenous (where each individual agent was evolved seperately and had its own unique behaviour algorithm) and pseudo-homogenous (where agents were evolved in "squads" of two or three players, with the same algorithm controlling each player in any given squad) evolution strategies, and found that, in competition, the heterogenous teams generally beat the pseudohomogenous ones. This indicates that allowing agents to specialise their behaviour may result in better performance. Aronsson's heterogenous agents managed, to some extent, to get past the kiddie soccer suboptimum, as some players exhibited defensive characteristics, ie they would not attempt to run after the ball, but instead wait between the ball and the goal until the ball was kicked towards them.

Aronsson's approach is unique among those described in that it tries to evolve individual agents, rather than a whole team. It is interesting that, despite this, the agents still showed some rudimentary teamwork. It is hard to tell whether evolving entire teams, individual players or small squads will prove the most effective strategy. Evolving individual players may lead to a more human-like result, as players specialise and become better at some skills than others, like human players. However, evolving a team of homogenous agents could allow for more flexibility; any agent would be able to perform any skill and play in any position with equal effectiveness, meaning the agents could swap positions at will during the game. The main question here is whether or not these more versatile agents would be as capable as their specialised counterparts, and if not, whether they would still be sufficiently good as a team to overcome their individual shortcomings. This touches on a debate from human football: the importance of the team's cooperative ability versus the importance of individual skill, and whether it is worth selecting players who are (for example) individually brilliant and possess incredible skills in attack, but possess little positional awareness or tackling ability and therefore contribute little or nothing to the team in defensive areas.

Chapter 3

Preparation

Before I could lay down a design for the framework, there were several preparatory steps which I needed to take to ensure that the design would be realistic, efficient and useable. Sections 3.2 and 3.3 precisely define the role of the module that I was trying to evolve and the parameters that the evolutionary process would need, while section 3.1 describes the process of selecting off-the-shelf code for use in the project implementation.

3.1 Toolkit Selection

Because this project involved several different components (the soccer server, the agents to evolve and the GP mechanism), all of which would need to be linked together, I would never have been able to even attempt it if it were not for the fact that there was a reasonable amount of pre-existing, open source code available. Trying to learn about and implement even one of the components listed above would probably have been too much for a final year project, so this pre-existing code was very important. However, although I did not have to try and implement an entire RoboCup team or GP mechanism, I still had to try and understand all of the code and then link them all together, when they were designed completely independently and bore absolutely no relationship to each other (this particular challenge is discussed at length in this report; see especially sections 4.1 and 5.3, and chapter 6).

This section details the best of the toolkits available in the areas which were useful to me: open source RoboCup teams and GP toolkits. The soccer server with which these toolkits would need to interact is described in section 2.1.2.

3.1.1 RoboCup Teams

Although binary executables are (theoretically) available for all teams competing in the RoboCup tournaments, the number of developers who choose to release their source code publicly is fairly small. The following list represents teams which have had their source code released after achieving success in a RoboCup tournament (ie teams which can reasonably be presumed to actually be good):

- CM United 1999 [27]
- UvA TriLearn 2003 [11]
- Brainstormers 2005 [20, 21]

3.1.1.1 CM United

CMUnited was one of the first teams to be publicly released and, to the best of my knowledge, was the first to be commonly used by developers looking for a package enabling them to dive straight into the artificial intelligence aspects of creating RoboCup agents. It was developed at Carnegie Mellon University in the USA, and was the winner of RoboCup '99. The team scored 110 goals over the course of the tournament (in which they played 8 games) without conceding any.

The package contains very little except methods automating communication between the client and the server. The developer must directly use the soccer server's basic `kick`, `turn` and `dash` commands when formulating behaviours, rather than being able to use higher-level methods which utilise combinations of these methods.

3.1.1.2 UvA TriLearn

The TriLearn team is the product of the Universiteit van Amsterdam. The source code released is the 2003 incarnation of the team, which in that year won the American Open, the German Open and the RoboCup world championships. The team's performance in RoboCup 2003 saw them score 177 goals and concede only 7 in 16 games.

The released code is not complete; the team's high-level decision-making module is not present, with the agents defaulting to the simple kiddie soccer strategy of "run towards the ball and kick it at the opponents' goal". However, the team's low-level (eg turning or dashing towards a specific point) and mid-level (eg passing, dribbling and marking an opponent) skills are all included in the released code. All the mechanisms for communicating with the soccer server and building a model of the agent's environment (a *world model*) from the information received are also included. The release is intended to allow developers to bypass the coding of actual skills, and allow them to focus instead on the processes involved in deciding which of these skills to use.

Alongside the player implementation, there is also a coach implementation which uses largely the same classes (eg for listening for information from the server and for building a model of the world).

3.1.1.3 Brainstormers

Brainstormers is developed at Universität Osnabrück, Germany. Their release is the most recent of the source packages reviewed, being the same code as was used by the Brainstormers team which won RoboCup 2005 and the 2005 German Open.

As with the TriLearn package, Brainstormers' decision-making module is not included in the release, instead being replaced by a simple default behaviour. The agents' skills are all present however, including passing, shooting, ball facing and searching and various types of interception and dribbling. As well as these relatively simple skills, the Brainstormers release also includes some more complex, co-operative skills involving multiple agents, such as dynamic agent positioning and scoring a goal from a specified position. A substantial number of these skills have been generated by the application of the developers' research into reinforcement learning using simulated neural networks.

The package also contains the Brainstormers' online coach, SputCoach.

3.1.1.4 Evaluation of RoboCup Teams

The CMUnited package is very basic, which is not really surprising given its age. It includes only the lowest-level skills, leaving the user to implement anything more than this. I decided that in my limited timeframe, this would be highly impractical; the time available during a final year project is far too short to contemplate attempting to create methods for such simple skills as dribbling and shooting at goal. With both the other packages, this work is already done. I judged that even if it were to turn out that I did not need the higher-level implementations provided in the TriLearn and Brainstormers releases, I

should still be able to simply not use them. In this case, it would be far easier to have something ready-done and then not use it than to not have something and have to create it myself. For this reason above all others, the CMUnited package was never a strong candidate for use in the project, despite the fact that I had previously used the package for AI22 coursework when I took the module in 2003.

TriLearn and Brainstormers, on the other hand, both contain a reasonable selection of pre-implemented skills. Although Brainstormers' are more extensive than TriLearn's, TriLearn's skill set covers all of the bases I expected to need. Both packages include a dribbling skill; given that I hope to try to evolve a dribbling skill myself as a test before trying to evolve multi-agent behaviours, this inclusion will be useful because it will give me a benchmark against which to test my evolved skill. Neither package provides a keepaway implementation however, so I would need to use a different benchmark such as Di Pietro's evolved keepaway agents (see section 2.1.5). In this respect, the fact that Brainstormers provides a larger skill set than TriLearn was not really an issue, as the added skills were not ones which I expected to need anyway. However, having the lower-level skills pre-implemented was definitely a plus as it allowed me the comfort of having a backup if my evolved skills turned out poorly. If my attempts to evolve the dribbling skill were to fail to produce a satisfactory result (and from the literature I knew this to be a very distinct possibility), I would still be able to attempt the evolution of keepaway agents using the pre-implemented skills.

Apart from the skills provided, Brainstormers and TriLearn present pretty much the same functionality. However, I found a distinct differnce between the two in terms of their coding. The TriLearn implementation benefits from having a very clear and strongly-imposed structure, seperating the different levels and modules of the agent's behaviour. The structure used is (to me, at least) very intuitive, with the release divided into the following main modules (which I have named after their chief component classes):

- BasicPlayer, which includes the low- and mid-level skills included with TriLearn;
- Player, which does the agent's decision-making and generates an appropriate command for the agent to perform;
- WorldModel, which represents the state of the world as the agent perceives it;
- SenseHandler, which listens for sensory information from the server and then interprets it and adds it to the WorldModel;

- ActionHandler, which translates commands into the appropriate format for the server and handles the sending of these commands.

The Brainstormers package, by contrast, is far less well defined. The world model and communication functions are seperated out from the agent behaviour, but I found that the seperation within these domains was far less obvious, making the package as a whole much less easy to understand and, in all probability, to work with. I was especially concerned about the skills provided by Brainstormers. As mentioned above, I was expecting to only want to use a subset of the behaviours provided by the packages. In the TriLearn, the skills in BasicPlayer are themselves placed in a three-level hierarchy: a skill can call upon any of the skills lower than itself (eg an interception skill could make use of a dash-to-point skill), but not those on equal or higher levels. The Brainstormers agent's skills could also be termed as low- or high-level, but no strict hierarchy is implemented; any skill can make use of any other. This is best explained in the developers' own words ([21], page 3):

> "...there is no strict hierarchical sub-divisioning. Consequently, it is also possible for a low-level behavior to call a more abstract one. For instance, the behavior responsible for intercepting the ball may, under certain circumstances, decide that it is better to not intercept the ball, but to focus on more defensive tasks and, in so doing, call the "defensive behavior" delegating responsibility for action choice to it."

While this feature may well be useful for creating the best possible agents, it made my position more difficult, as I could not be sure (without lengthy manual checking) which of the pre-implemented behaviours my agents were actually using. I did not want to be in the situation where I was trying to evolve good dribbling behaviour, but I was actually calling the pre-implemented dribbling method without realising it. With TriLearn's rigid hierarchy, this would be easy; with Brainstormers it would be much harder.

Brainstormers' high level of interlinking also made the code far less easy to understand. With TriLearn I was able to very quickly understand how the different elements ofthe implementation fitted together, how I should go about building an agent, and how I might need to alter the structures provided in order to include the GP. this was not the case with Brainstormers. I perceived the learning curve involved in using TriLearn to be far gentler than that of Brainstormers, and this perception was reinforced by the knowledge that there were people within the School of Computing (specifically my supervisor Dr. Bulpitt) who were familiar with the TriLearn package. I knew that if I used TriLearn, then I could

rely on expert backup even if I did have problems, whereas I would not have the same level of support if I were to choose the Brainstormers package.

Because of the above evaluation, I decided to use the TriLearn package for my project. For future reference, the skills which are provided with the TriLearn package are listed in Appendix B.

3.1.2 Genetic Programming Toolkits

The selection of a GP toolkit was restricted by the requirement (see section 2.2.2) that it be written in C++. After surveying the available toolkits, it became apparent that there were only two viable options:

- Open BEAGLE [9]
- EO Evolutionary Computation Framework [10]

3.1.2.1 Open BEAGLE

OpenBEAGLE describes itself as a "versatile EC framework [30]". It is a high-level framework, and was first developed at Université Laval, Canada, with additional contributors from other universities in the US and New Zealand. It has inbuilt support for all the main varieties of EC, including GP and STGP, and includes implementations of commonly used versions of these (eg. bit-string and real-valued GAs). Open BEAGLE uses strong object-oriented principles and polymorphism to allow easy code reuse. The package also makes heavy use of XML, using the format not only to write and read configuration files and logs, but also to control, at the highest level, the evolutionary process itself.

3.1.2.2 EO Evolutionary Computation Framework

EO's development began at the University of Granada in Spain, and the development team now also includes members from universities in France and the Netherlands. EO makes extensive use of templates to allow the user to perform evolutionary operations on any data-type; all the operations provided in the package are templatised and thus can be applied to any data-type implementing two specific methods. This makes the package highly flexible and means that the user is not restricted to any particular EC paradigm; the user has complete freedom over how to construct their evolution. To this extent, EO actually supercedes the standard EC paradigms, although all of the standard four are implemented and ready to use, as is STGP.

3.1.2.3 Evaluation of GP Toolkits

My decision was that I should use the EO framework, due to its flexiblity and the fact that it is a much smaller package than BEAGLE. Due to the nature of my project and what I was attempting to evolve, I felt that it may well be necessary to use complex fitness functions and mutation operators, and I expected EO's flexibility to make implementing these much more straightforward. Also, I felt that the relatively large size of the BEAGLE package was intimidating. In a project of very limited time, I didn't want to be wading through large quantities of irrelevant code, and nor did I want to be burdened with an overly complex toolkit with lots of features that I didn't need and would simply get in the way.

Despite these seeming advantages, choosing EO turned out to be a mistake, as shall be seen in section 5.1.1.

3.2 Role and Location of the Evolved Module

It was all very well saying that I wanted to evolve agents which could dribble and agents which were successful at keepaway, but what did that actually mean in practical terms? Having decided to use TriLearn as the basis for my agents, I could now work that out. With TriLearn providing the skills that I needed my agents to use, what I needed to evolve was the decision-making module which would decide which of the skills to use and when to use them. So, I first had to locate the part of the TriLearn release which is the agent's decision-making module.

The agent included with TriLearn is started by running the executable `trilearn_player`. Running this executable once starts one agent and attempts to connect it to the soccer server (thus the soccer server must be started *before* `trilearn_player` is run). A seperate script (start.sh in the TriLearn directory) is included in the release and can be used to run `trilearn_player` multiple times, thus starting a team of agents. By default, start.sh starts an entire 11-agent team and an online coach, although this can be altered using command line parameters.

Looking at the main function corresponding to `trilearn_player` (in file src/main.cpp), it can be seen that the main function creates an object of the class `Player` and then calls the method `Player::mainLoop()`. This method updates the agent's world model if there is new sensory information available, and then calls another function depending on what type of player the agent is (defender, midfielder, attacker or goalkeeper). These methods are the agent's intelligence, where the decisions are made as to what actions the agent should take, and so it was obviously here that my evolved mechanism

would need to go.

3.3 Specification of the GP Parameters

Obviously, I was going to use the toolkit to do most of the evolving work. However, I still needed to define the key GP elements for this particular problem.

3.3.1 Primitives Set

Outputs

In defining my primitive set I was faced with a series of choices. The first of these concerned exactly what output I wanted my tree to be giving. Previous experiments with using GP for RoboCup have used the soccer server's inbuilt commands (kick, turn, dash etc) as the outputs for their GP trees. However, using the TriLearn team as a base meant that I had the option of outputting the skills from TriLearn's `BasicPlayer` class (`dashToPoint`, `turnBodyToPoint`, `tackle` etc). This second approach would allow me to operate at a higher level of abstraction, which I thought would be beneficial to my results; Wilson's experiments with different primitive sets got the best results when more complex skills such as `kickTo` and `hasBall` were included as primitives (see section 2.3.2). Because of this, and because of the fact that it would be breaking new ground rather than retracing others' steps, I chose to abstract away from the soccer server's basic commands and use TriLearn's skills as outputs from my tree.

This led naturally on to deciding which of the TriLearn skills I should use (see Appendix B for a list of TriLearn's pre-implemented skills). I decided to limit myself to the skills which the TriLearn developers describe as "low-level", as I anticipated that the more complex I made my primitives, the more complex I would make the implementation as a whole. Using these more complex primitives is something that may be worthwhile as future research.

My eventual list of output primitives consisted of:

```
turnBodyToPoint
turnNeckToPoint
alignNeckWithBody
dashToPoint
searchBall
```

`freezeBall`

`accelerateBallToVelocity`

`kickBallCloseToBody`

`tackle`

The skills which I left out (apart from all the "intermediate" and "high-level" ones) were `turnBackToPoint`, which I viewed as being too similar to `turnBodyToPoint` to warrant inclusion; `catchBall`, because this is a goalkeeper-specific skill and I was interested in evolving outfield players; `teleportToPos`, as this is a command which is only usable in certain situations (ie when the soccer server is in the `before_kick_off` state), and I felt this might overcomplicate matters; and `communicate` and `listenTo`, because I didn't plan to implement communication between the players (such communication may well prove to be beneficial to future implementations, but again I felt it would be overcomplicating matters to include it in mine straight away).

Inputs

Next I needed to decide what information it would be useful for the agent to know. This information would be input to the GP tree in the form of terminals, and used as arguments to functions until an output was reached at the root node. After looking back at previous research, I decided on the following list:

The position of the agent on the pitch

The velocity of the agent

The position of the ball relative to the agent

The speed of the ball

The direction of the ball

The position of the opponents' goal relative to the agent

The position of the agent's team's goal relative to the agent

Whether or not the agent can reach the ball quicker than any of its teammates

Whether or not the agent can reach the ball quicker than any other player

Whether or not the agent is in a position fo kick the ball

Objects representing all the other players on the pitch

Operators

As well as the inputs and outputs, I needed operators (functions) to link the two together. I found this group the hardest to decide on, as the functions would mostly be mathematical in nature and I wasn't sure which would be useful in the tree and which would not; it was not like (eg) the inputs where it was relatively straightforward to say that, for example, knowing the position of the ball was probably going to be more useful to the agent than knowing the position of the corner flag. After reviewing the literature, I decided that the following set would be a good starting point:

Standard mathematical operators (addition, subtraction, multiplication and division)

Comparison operators (`equalTo` and `lessThan`)

`ifThenElse`

`getPlayerPosition`

`getPlayerVelocity`

The mathematical and comparison operators would work exactly as would be expected, as would the `ifThenElse` function. A function such as this one has been included in all of the RoboCup-GP implementations that I have read about, and it seems obvious to me that this operator would add a lot more power to the tree. To briefly sketch its implementation, `ifThenElse` would be a three arity primitive, taking a boolean value as its first argument and any type of value as its second and third. It would return either its second argument or its third, depending on the value of the boolean argument.

The two other functions, `getPlayerPosition` and `getPlayerVelocity` would be operators which would each take a single argument: one of the player objects mentioned in the inputs section. The function would then return the appropriate position (relative to the agent) or velocity.

There are doubtless many other primitives which I could have included, but I was very aware that every extra primitive increases the space through which the GP algorithm has to search. Because of this, I wanted to try and keep the number of primitives small. As it is, I may have found (had I had the chance) that even this number of primitives was too many, and the search space increased to the point where it was impossible to find a good solution. However, that is a question which I never had the opportunity to answer.

3.3.2 Fitness Function

As previously mentioned, defining a fitness function is often the hardest part of a GP implementation, especially for highly complex problems such as RoboCup. When trying to evolve full teams, there are several methods which have been tried by researchers in the past: Luke [16] simply played matches between his evolved teams and took whichever team won the match to be fit and the other not to be; Andre and Teller's Darwin United [1] used a complex scoring system to evaluate their teams, where a team would get fitness points for winning the game, scoring goals, kicking the ball, keeping the ball in the opposition's half etc. The points awarded for these different accomplishments were scaled so that the more significant achievements (eg scoring) scored far more fitness points.

Both approaches have their disadvantages. Luke's tournament selection has the potential for inferior solutions to be selected simply because they scored a lucky victory over a superior opponent, and the Darwin United approach can be criticised because the scoring system may be biased towards teams which play in a certain way, which will reduce population diversity and lead to premature convergence.

This difficulty in selecting a fitness function is the main reason why I decided on the two scenarios that I did: both dribbling and keepaway have relatively simple measures of performance which can be used as fitness functions. Keepaway is especially simple, as the fitness function would simply be the amount of time the keepers can prevent the takers from winning the ball. A dribble skill would be marginally more complex, but still workable. The elements which need to be taken into account in defining the performance of a dribbling player would probably be the speed of the dribble, the average closeness of the ball to the player and the accuracy of the dribbling (ie did the player actually dribble in the right direction). The dribble speed and accuracy could be measured by the distance from the start point to the end point of the dribble divided by the time it took to get there, and an the average closeness of the ball could be measured by how many times the ball was kicked (the more kicks, the closer the ball stayed to the player). The aim of the process would be to maximise both of these, giving a fitness function of $\frac{distance}{time} \times kicks$. The higher the agent's score, the fitter the agent.

3.3.3 Termination Criterion

I decided that the termination criterion should, initially at least, simply be a number of generations. Knowing how many generations will be enough for the programs to achieve competency is always a bit of a guess in GP. Looking back at the reports of other researchers doing work on GP for RoboCup, Luke

used a total of 52 generations for U.Maryland [16] (although this would have been more had Luke not manipulated his population specifically to try and reduce the number of generations) and Wilson used 32 for his experiments [34].

These are very small numbers compared to most GP applications, where the number of generations is often counted in thousands (as mentioned previously, Koza recommends 4000 generations to be a reasonable estimate, although it is acknowledged that the numbers can differ significantly depending on the problem at hand and the setup of the GP). The reason for these very small numbers of generations is that evaluating each generation of agents takes a very long time, as each agent in the population must take part in at least one match in order to be evaluated; preferably more than one to give increased robustness of results. Given that a full RoboCup match takes ten minutes, evolving a population of a worthwhile size (Wilson had 256 individuals per generation, Luke between 100 and 400) takes a very long time. In Wilson's case, using tournament-style selection, a full run would have taken the better part of two months without steps being taken to speed up the process. Such steps are possible; the most common ones are reducing the length of each evaluation (match) and by processing multiple matches in parallel. Even allowing for this however, evaluation times are still prohibitively large.

Because of these factors, I decided that my initial termination criterion should be 30 generations. I deliberately set this low so that I could examine my initial results and see whether or not additional generations would improve them, and then run them if necessary. I felt this was preferable to running an extra 20 generations and then finding out that the time they took to run was wasted.

3.3.4 Other Parameters

Both of the GP toolkits that were considered (see section 3.1.2) included a selection of commonly used primitives, fitness evaluators and other GP variables. For the primitives and fitness function, it was obvious to me that my needs went beyond anything that would be supplied in a generic toolkit, and so I needed to implement them myself; however there were other parameters for the evolution for which one of the pre-implemented methods would be perfectly acceptable. These parameters are listed below.

Initialisation method

The two most basic initialisation methods are full initialisation and growth initialisation. In both methods, the trees are generated randomly; the difference between the two is that in full initialisation, all the tree's branches will be as long as they are permitted to be; in growth initialisation, the branch

lengths are random, up to the specified limit (see section 2.2.1 for brief notes on branch depth limits). Commonly, these two methods are combined in the form of ramped half-and-half initialisation, where each tree has an equal chance of being created using either of the above methods; this is the method that I chose to use.

Selection method

The most typical selection strategy in GP is fitness proportional, ie the probability of an individual being selected is directly related to how its fitness score compared to the fitness scores of the other individuals in the generation. The probability is calculated as the fitness score of the individual divided by the sum of the fitness scores of all the individuals in the generation.

Despite its popularity, there are recognised problems with this method. One is that a if few individuals in the population are significantly more successful than the rest, they can very quickly take over the entire population and convergence on a solution before the evolution has had much chance to explore the search space. In such scenarios, the solution arrived at is generally suboptimal. Another pitfall of fitness proportional selection is that it can lead to a lack of selection pressure when all the members of a generation have very similar fitness scores. In this case, there is little selectorial advantage gained from slight improvements in an individual's genome, causing the population to only converge very slowly.

I had no way of knowing whether either of the problems mentioned would affect my evolution. For this reason, I decided that I might as well use fitness proportional selection; it was implemented in the GP toolkits and was not going to cost me anything if it didn't work. I also knew what problems to look out for, which could help if I needed to try to diagnose problems with my setup later on.

Chapter 4

Design

4.1 Framework Design

As I was using toolkits, I already had modules which I could use, with a few modifications, as "black boxes". The main issue was going to be how I linked these different boxes together.

Figure 4.1 shows the design at the very highest level. The arrows in the diagram represent the information flows through the program as it runs; these numbers refer to the list below which charts the process of an evolutionary run from start to finish (Note: It is assumed that the soccer server is constantly running throughout).

1 The program is run, starting the central control mechanism (CCM)

2 The CCM calls the evolutionary mechanism (EM), causing it to initialise the population;

3 The CCM calls the coach and player agents, starting them; they in turn connect to the soccer server;

4 The CCM calls the EM to start the evolution;

4.1 The EM sends the GP tree to be tested to the CCM, which forwards it on to the player agent;

4.2 The EM sends a message to the CCM stating that it is ready for the tree to be evaluated; the CCM forwards this message to the coach, which moves the connected players to their starting positions on the pitch (the server automatically informs the players of this);

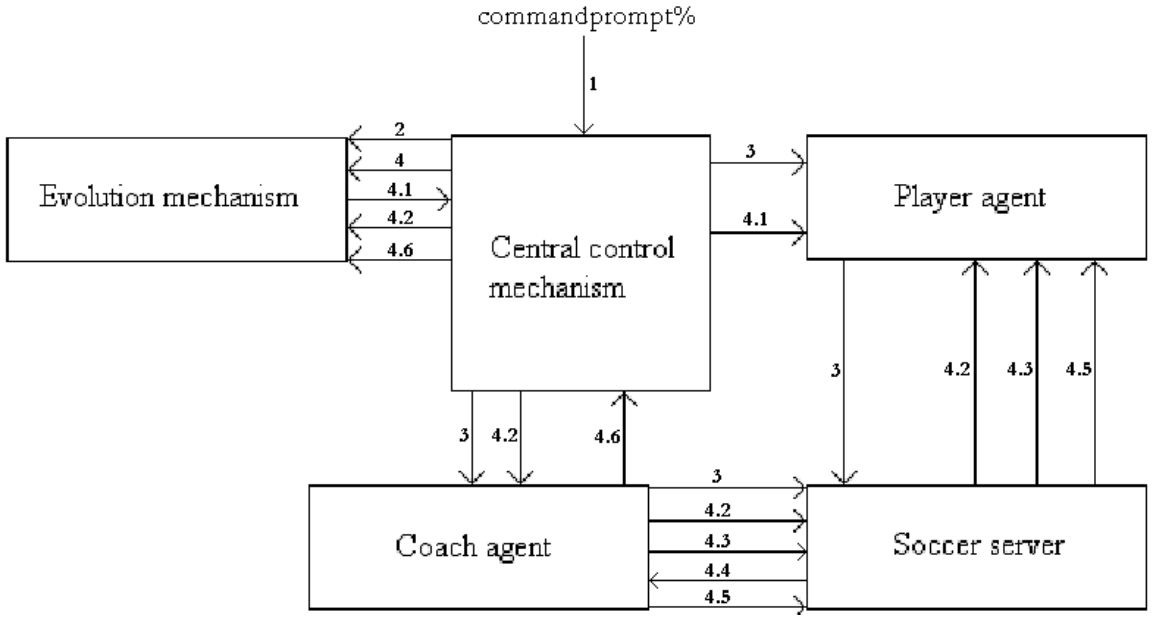

Figure 4.1: Top-level view of the framework design and the flow of information through it

4.3 The coach starts the evaluation match (the server automatically informs the players of this);

4.4 The coach monitors the match and keeps it running until the evaluation is finished;

4.5 The coach stops the match (the server automatically informs the players of this);

4.6 The coach sends fitness information back to the CCM, from where it is forwarded to the EM.

Steps 4.1 through 4.6 are repeated for all the other individuals in the generation. At the end of the generation, the EM performs selection and recombination on the individuals, and then starts again with the evaluation process (4.1 through 4.6) for this new generation, continuing until the maximum number of generations is reached. At this point, the EM signals to the CCM that it has finished, and the CCM tidies up by shutting down the players and coach.

Two things should be pointed out about this design. Firstly, that the evolution mechanism is kept completely seperate from the agents. This is important; even though the evolution was going to be operating on the player agent, once it was evolved this agent would have to be able to operate on its own, eg in a normal game. In this situation, it would merely be a hindrance to have to keep the evolution mechanism together with the agent. Secondly, all significant communication with the soccer server is passed through the coach. Although the control and/or evolution mechanism could theoretically need to change the state of the server, only the coach and player can actually connect to (and therefore

communicate with) the server. As the coach has more control over the server than the players, control of the evaluation is devolved from the evolutionary mechanism to a specially designed coach. No modifications to the server are necessary, except possibly for the alteration of some parameters to make it run faster (as described in section 3.3.3).

It is immediately obvious that the CCM is critical to this framework. The following section describes the CCM in more detail.

4.1.1 Central Control Mechanism

The key component of the CCM is the executable which is run to start the whole evolutionary process; steps 2, 3 and 4 will be directly initiated by it, at which point the EM can take over directing the proceedings. However even at this point, the CCM will be important as a conduit between the EM and the agents. This functionality will not be implemented within the executable, but rather as a seperate class `FYPIntegrator` which can be directly used by both the agents and the EM.

4.2 Implementation Plan

4.2.1 Initial Plan

4.2.1.1 Phase 1: Implementing GP Base

The first part of the implementation was to create the necessary environment for the GP package to function properly. This involved the following work:

- Implementing the GP primitives defined in the preparation (see section 3.3.1);
- Laying down the basis of the custom fitness evaluator (to be completed in phase 3);
- Creating a basic CCM, ie capable of initialising the EM with the parameters defined in the preparation (section 3.3).

4.2.1.2 Phase 2: Implementing Agent Base

Phase 2 involved preparing the TriLearn package for use with the CCM, specifically:

- Implementing a player agent to use the evolved GP trees;
- Implementing a coach to control the evaluation of the player trees;

- Adding functionality to the CCM to allow it to start and appropriately configure the agents;
- Altering the soccer server parameters to values appropriate for use with the system.

4.2.1.3 Phase 3: Linking the Evolutionary Mechanism and the Agents

Phase 3 of the implementation would see the final joining together of the agents and the EM:

- Completing the CCM.
- Completing the evaluator from Phase 1;

4.2.2 Revised Plan

This initial plan was drawn up when I intended to use the EO framework for GP. After I was forced to change to using BEAGLE, I reviewed the design and the implementation plan and decided that the design and implementation plan were still viable, with the single exception that the following requirement needed to be added to Phase 1:

- Creating an XML file to appropriately direct the evolution (a "custom evolver" in BEAGLE-speak).

Chapter 5

Implementation

5.1 Phase 1

5.1.1 Implementing the Evolution Mechanism using EO

Initially, I attempted to implement the Evolution Mechanism using the EO library. However, this attempt got precisely nowhere. After struggling with the library for over a fortnight, I came to the conclusion (in conjunction with Dr. Bulpitt) that the library simply wasn't usable, for the following reasons:

- The documentation provided with the library was far poorer than it appeared at first glance. When I made my decision about which library to use, I examined EO's documentation (whose chief components are an online tutorial [32] and the interface documentation [31]) and decided that it would be usable; I found it to be reasonably clear and easy to follow. However, when I came to actually implement the system, I found that the main reason that I could follow it so easily was that it only provided a description of how to implement a very simple GA, and did not actually cover the mechanics of implementing a complex GP system.

- The library's source code was almost impossible to read, mainly due to its pervasive use of templates. While C++ template classes and functions have their advantages, improving the legibility of the code is not one of them. This meant that not only was there no useful documentation to

help me, but I couldn't use the source code to try and fathom how the library worked either.

- The library's structure was not intuitive to me; I found it very difficult to work out how the different pieces of the library were supposed to fit together, and the documentation could not give me any pointers either.

The upshot of these three facts was that although I could quite easily have implemented a simple GA using EO, I had no way of knowing where to start implementing the complex system that I was trying to create. After much consultation with Dr. Bulpitt, we decided that the only viable path was to give EO up as a bad job and try using another library.

5.1.2 Implementing the Evolution Mechanism using BEAGLE

To allow for relatively easy implementation of customised evolutions, BEAGLE provides abstract base classes for all aspects of the evolutionary process. To create custom versions, it is simply necessary to subclass these abstract classes.

5.1.2.1 Implementing the Primitives

Data types

(See appendix C for a brief description of the custom TriLearn data types referred to in this section)

Before I could get to implementing the primitives themselves, there were several steps which I had to take to allow BEAGLE to use the data types which I needed. The TriLearn skills which I chose to use as primitives require some custom data types, namely `VecPosition`, `AngDeg` and `SoccerCommand`. I had also defined a primitive representing a player, for which I intended to use TriLearn's enumeration `ObjectT`. I could not simply put these into BEAGLE and expect it to work though. BEAGLE does not directly use simple types; Instead it uses instances of a wrapper class `Beagle::WrapperT<T>`, which contain (and are templatised on) primitive types. For instance, BEAGLE does not directly use `int`s; it uses `WrapperT<int>`s, which are then `typedef`'d to the name `Integer`. Following that lead, I `typedef`'d all the wrapped custom data types as well by appending a "W" to their name, eg `SoccerCommandW` was an alias for `WrapperT<SoccerCommand>`.

For the `WrapperT` class to function, the wrapped class must explicitly define a selection of overloaded operators, including < , == , << and >>. My first task, therefore, was ensuring that my data types were compatible with BEAGLE by implementing these operators, as most of them were not needed, and

therefore not implemented, in the TriLearn package itself. Implementing these operators threw up one or two rather bizarre questions, such as how can one `SoccerCommand` object be "less than" another? To my mind, the answer to this is that no command is less or more than any other; however, this oddity of BEAGLE's architecture forced me to try and define a way of ranking them.

Primitives

In BEAGLE every primitive is implemented as a seperate class, and all primitive classes inherit from the abstract base class `Beagle::GP::Primitive`. Fuctions inherit directly from this class; terminals inherit from a direct subclass of `Primitive`, `Beagle::GP::TokenT<T>`. `TokenT<T>` is templatised with respect to the return type of the terminal.

To instantiate the abstract class `Primitive`, a primitive class must implement the method `virtual void execute(Beagle::GP::Datum&, Beagle::GP::Context&).` This is where the primitive carries out its designated operation. For example, in the terminal primitive `FYPWorldModelInfoGetBallPos`, the `execute` method retrieves the ball's position from TriLearn's world model and outputs it; in the function primitive `FYPDashToPointOutput`, it takes the arguments it recieves, constructs a `dashToPoint` command based on them, and outputs the command.

This one method is sufficient for GP, but not sufficient for STGP. For STGP, it is also necessary to implement the methods `virtual const std::type_info* getArgType(unsigned int, Beagle::GP::Context&) const` and `virtual const std::type_info* getReturnType(Beagle::GP::Context&) const`. These methods perform the job of ensuring that STGP type constraints are not violated.

The primitives were, despite their apparent simplicity, surprisingly time-consuming to implement. For a start, there were simply so many of them: given that using STGP meant that I had to define seperate operators for each different data type (ie I had to define one `FYPLessThan` operator for comparing two `VecPositionW` objects and a completely seperate one for comparing two `ObjectTW`s, etc) I had to implement a total of 36 primitives. There were also extra complications involved in trying create terminal primitives: because the base class for terminals, `TokenT<T>`, is templatised, I had to jump through some extra hoops to make sure that the terminals were implemented in such a way that they would still compile. For instance, the templatisation meant that terminal functions had to be not just

defined, but also implemented in header files, not implementation files. Trying to get the templates to work with instances of `ObjectTW`, which is an enumeration as opposed to a class, was particularly trying. In the end, I was forced to subclass terminals involving `ObjectTW`s directly from the class `Primitive`, rather than from `TokenT<T>`; this violated the structure of BEAGLE's implementation of primitives, but I was at a loss to find any other way of making it work.

5.1.2.2 Creating the Structure of the Fitness Evaluator

The BEAGLE base class for GP evaluation operators is `Beagle::GP::EvaluationOp`. The method which must be implemented for subclasses of this is `Beagle::Fitness::Handle evaluate(Beagle::GP::Individual&, Beagle::GP::Context&)`. The purpose of this method is obvious; it runs the actual evaluation of the individuals.

At this stage, all that was necessary was to create a subclass of `EvaluationOp`, `FYPEvalOp`, with the `evaluate` method implemented as a stub.

5.1.2.3 Creating the BEAGLE Custom Evolver

A BEAGLE custom evolver is simply an XML structure which tells the BEAGLE structure which versions of the evolutionary operators to use. It is part of the BEAGLE configuration (.conf) file. The .conf file for this project is reproduced in Appendix D.

5.1.2.4 Creating the basic Central Control Mechanism

At this stage, all that was necessary in terms of the executable file was that it would run BEAGLE, therefore I could get away with using one which was modified only slightly from the examples in the BEAGLE GP tutorial [29] and release's examples directory. The only changes necessary from the examples were to change the included primitives and to specify that my custom evaluation operator should be used.

I also needed to create the `FYPIntegrator` class, mentioned in section 4.1.1. This was necessary because a large number of the primitives which I implemented above were based on methods from the TriLearn package, and as such I needed a structure in place to enable them to use it. I achieved this by allowing the `FYPIntegrator` class to contain a reference to the same TriLearn world model as would be used by the player agent, and allowing the primitives to access this reference. The reference had to refer to the world model used by the player rather than that used by the coach because the two models

would be different: the coach's world model would not have the noise added to it that the player's would have. As I was attempting to evolve a controller for the player rather than the coach, it would obviously not have been appropriate to have used information from the coach's world model to evolve the player.

5.2 Phase 2

5.2.1 Implementing the Player Agent

Unlike BEAGLE, TriLearn is not specifically designed to be flexible and easy to customise; however as there was not too much that I wanted to change about the player agent, making the necessary changes was not too painful. In TriLearn, the agent's intelligence is implemented in the class `Player`, which is a subclass of `BasicPlayer`. I was able to simply subclass `Player` further to create `FYPPlayer`, and then override `Player`'s position-specific main loop methods (see section 3.2). As player positions had no relevance to either the dribbling or keepaway scenarios, I was able to have these position-specific methods call a universal `keepawayMainLoop` method; this meant that no matter what type a player might be defined as, it would still exhibit the same behaviour, which is precisely what I wanted.

5.2.2 Implementing the Coach Agent

This was implemented in almost identical fashion to the player agent. `FYPCoach` was defined as a subclass of TriLearn's `BasicCoach`, with the method `mainLoopNormal` overridden to allow me to implement the coach behaviour which would control the matches. In `mainLoopNormal`, I needed to try and define the mechanisms which the coach would use to control the evaluations. The stages referred to in this section are those used in section 4.1.

I began by trying to implement the controls necessary for evolving the dribbling skill:

Stage 4.2 (1) Generate a random location within the boundary of the pitch and send a command to the server to move the player there; (2) Select a second random position on the pitch, at least 20 metres away from the player and set this point to be the target for the player to dribble to; (3) Calculate and store the distance between the player and the target; (4) Generate a third random position within 2 metres of the line and between 2 and 8 metres from the player, and send a command to the server to move the ball to this position.

Stage 4.3 Send start command to server.

Stage 4.4 Keep analysing the positions of the ball and the player; if either (a) the player and the ball both come within 1 metre of the target, or (b) the number of cycles passed since the starting of the server reaches 200, then ...

Stage 4.5 (1) Record the distance between the player and the target and the distance between the ball and the target (record 0 if the distance is less than 1); (2) Record the number of cycles passed since the start command was sent; (3) Send stop command to server.

It should be noted that all but one of the values required for calculating the agent's fitness have been recorded by the steps above, but I have not mentioned anything about the number of kicks performed by the agent. This is because the coach is not quite so omnipotent as it's made out to be; although it can see everything happening on the pitch, it can only record the positions and velocities of objects. It cannot monitor what commands the players have just carried out, and as such cannot count kicks. My solution to this problem can be found in section 5.3.2.

The way of managing the dribble evolution described in stage 4.4 meant that I needed to add an extra primitive to my set: a terminal representing the target point of the dribble.

Stage 4.6 was left to be completed in phase 3, as its implementation relied heavily on the mechanisms used to link the coach to the EM.

5.2.3 Updating the Central Control Mechanism to Start the Agents

At this point I needed to make some decisions about the structure of the executable that I was going to use to set the evolution running; I needed to ensure that all the information necessary for setting up the different components was available at the right time. It took some trial and error to work out the best way of doing this; my first thought was that it was be most logical to have the main executable set up the EM structure and then allow the EM to start the agents. My reasoning was that as the EM was the component which would guide the whole process, and would be largely controlling the actions of the agents, it made most sense to have the EM start the agents as well. However, this turned out not to be possible, for the reason that many of the primitives within the EM needed access to the player agent's world model; if the player agent wasn't created, then there was no world model for these primitives to access. That meant that my main executable had to first start a player agent, then start the EM using the world model from the player.

After this, I could then start the coach. However, because the coach had to create as a seperate

process and via a different port than the player, I needed to have a seperate executable for this, and use the C++ `popen` function to start that executable in a seperate process.

I could foresee a problem with this structure: how would it work when multiple player agents were needed? Each agent has its own model, and as each of them has access to different information about the world, each model will contain different information. But if I was going to have to link the evolutionary process so closely to one particular agent, then the EM would have no access to any additional player agents which I might then introduce in trying to evolve agents in a more complex scenario than simply dribbling the ball. As far as I could work out, running two agents simultaneously with the EM would result in both agents' decisions being based on the information from a single world model; effectively, all the agents would be seeing the world from the point of view of one.

As I was very tight on time at this point, I left this problem and moved on. However, I was forced to return to it in completing the CCM, as other issues with this implementation structure came to light. See section 5.3.1 below.

5.2.4 Tweaking the Soccer Server Parameters

As mentioned in section 3.3.3, it is possible to alter the parameters of the soccer server to make it run faster. This involved modifying the soccer server's configuration file to reduce certain intervals:

- The intervals between the execution of player action commands: 100msec to 20msec;
- The intervals between the server's acceptance of player action commands: 10msec to 2msec;
- The intervals between the sending of visual updates to the players: 150msec to 30msec;
- The intervals between the sending of proprioceptory updates to the players: 100msec to 20msec;
- The intervals between the sending of visual updates to the coach: 100msec to 20msec.

Other changes were required to make sure that the server did not automatically change its play mode at any point, an action which might have disrupted the agents' play during the evaluations. I needed to make sure that the coach had total control of the evaluations, and so the following parameters were altered:

- The length of a half was changed to -1, to prevent the server from automatically ending play after a certain amount of time;
- The length of the "drop time" was reduced to 0 (ie switched off); the drop time is the amount of time that a team is allowed to delay taking a set piece.

It should be noted that the tweaked settings were not used when testing the framework in phase 3, as I wanted there to be as few variables as possible which could be causing problems.

5.3 Phase 3

5.3.1 Completing the Central Control Mechanism

Up until this point, I had found the implementation difficult. However, phase 3 was when I found the problems began to spiral out of control. My first problem was that, as I had thought it might, the structure implemented in phase 2 turned out to be unworkable. This was even true for evolutions using only a single player agent, which I had thought might still be viable. As such, I had to revert to the more logical structure described in section 5.2.3: that of initialising the evolutionary process first. This meant that the initialisation of the player and coach agents could be moved back into the evaluation process, where they were originally planned to be in the design. Having the intialisation in the evaluation meant that the framework would create a new set of agents for every new evaluation and then destroy them when finished, rather than the previous setup of having a single set of agents open at all times, but pointing them at different trees when they were run for the evaluations. See section 5.3.2 below.

Having made this change, however, I came straight back to the problem mentioned before: that the world model must be created before the primitives in order for the primitives to have something to refer to. This problem had me completely defeated for a while, until my attempts at solving a different problem pointed to a potential solution. This problem was trying to work out how to send the GP tree to the player agent so that the player could run it during the evaluations; my investigations into this pointed to a possible solution to both problems. I shall describe the problem with sending the tree first, and then talk about my attempts at solving it and how they found a possible solution to the world model problem also.

In BEAGLE, an individual is run using the `run` method in the `Beagle::GP::Individual` class; to run the tree in the player agent, I needed to be able to call `run` from inside `FYPPlayer`'s `mainLoop` method. The problem with this was that the player was running in a completely seperate process to the EM, so simply passing a reference to the `Individual` through the system was impossible. Because of this, I was forced to investigate methods of sharing data between processes. For me, this was delving into unknown territory; my C++ knowledge was limited before the start of the project, and even after several months of using the language for the implementation I could not claim to have a

detailed knowledge of it. I had no knowledge at all of multi-process or parallel programming, and very little of detailed memory management. This all had to be learnt before I could begin to use or understand the techniques discussed below.

My first attempt at implementing data-sharing was using pipes. I was using the `popen` method to open new processes; as well as starting the child process, this method also creates a unidirectional pipe between the parent process and the child process. This pipe had already proved very useful in getting data back from the player and coach agents to the evaluation operator (see section 5.3.2 below). However, I needed bidirectional communication between the evaluator and the agents, and extensive attempts to create a second pipe between the two processes to carry data in the opposite direction proved fruitless. Although it was not too difficult to create a pipe in one process, it proved impossible to get the other process to recognise the pipe's existence, regardless of which of the two processes was the creator.

In much frustration, I was forced to consider another alternative: shared memory. Shared memory is exactly what it sounds like: memory which can be accessed by several different processes. Once a shared memory segment has been created, any process which possesses the correct key can access it. It sounded ideal; it also sounded complicated. On the second count I was definitely not mistaken; on the first, I couldn't say. By the time I had learnt enough about the concept and use of shared memory to be able to use it, I had insufficient time left to complete the implementation of it.

I believe that it would be possible to store the variables which multiple processes need to access, or at least references to those variables, in a shared memory segment. The variables affected would chiefly be the GP trees and the agents' world models. I did manage to get shared memory set up for the trees; I was able to place a reference to the tree in shared memory, then pick this reference up and use it in the seperate player process. After my previous experiences, I performed careful testing to make sure that the segment was actually working properly, but the test results convinced me that it was in fact working. This, I believe, was a major breakthrough; the only disappointment was that it came too late to save the implementation. Above all other things, it is the fact that I managed to get this shared memory to work in this aspect that encourages me to believe that the framework may actually be relatively close to completion. See section 6.6 for a more detailed analysis.

It should briefly be noted that my efforts at getting the different processes to work together were also hindered by a mysterious problem with the soccer server. When agents connect to the server, they connect to different ports depending on their type: players connect via port 6000, trainers via port 6001 and online coaches via port 6002. As stated previously, I was interested in using the trainer rather

than the online coach; however, attempts to connect any agent to the server via port 6001 failed. This was true even for the default TriLearn coaches, so the problem cannot have been a bug which was introduced by my code. Connecting TriLearn's coach in online mode via port 6002 worked fine, and trying to connect it to this port while in trainer mode failed and generated an error message, as would be expected. However, trying to connect the agent to port 6001 failed with no error message, regardless of the mode used. The agent sent the connect message but received no reply to it; the server gave no sign that it had actually received the connection request. My best guess as to the problem was that for some reason the School of Computing machines were blocked from using port 6001, as this could explain why the connection was failing silently. However, support was unaware of any such restriction, and so the origin of this problem remains a mystery to me.

5.3.2 Completing the Fitness Evaluator

With the design revised so that the agent processes were to be opened by the evaluation operator, the first thing to do in the evaluation operator was to implement the create the code to open them. As mentioned above, my research suggested the `popen` function to be the way to do this. The pipe created automatically as part of the `popen` method (see above) proved hugely useful, as it enabled the agents to send messages back to the evaluator without having to do anything particularly complex. At the child process's end, all I had to do was to send a message to standard output, which the `popen` function had rerouted so that instead of printing to the terminal, it was sent to the pipe into the parent process. In the parent process, it was a matter of listening to the pipe and parsing anything that came through it. I quickly knocked up a very basic system of messages for the player and coach to relay back to the evaluator; with more time and consideration such a system could be developed into an efficient mode of communication.

The main reason for wanting to get data back from the agents was to enable the calculation of fitness. At the end of the evaluation, the coach could send an appropriately phrased message containing the fitness data to its standard output, and the evaluator would pick it up. The same process can be used for player agents. It was mentioned in section 5.2.2 that there were some things that the coach can not pick up on, such as what actions the players have taken. With the pipe in place, the players themselves can send information about their actions directly to the fitness evaluator, where they can be processed appropriately (eg the number of kicks can be counted, to take the example of the dribble fitness function).

Chapter 6

Evaluation

6.1 Evaluation Criteria

Before the project can be evaluated, it is necessary to define the criteria against which it is to be judged. The first step in defining these criteria is the project's minimum requirements (see section 6.2 below); however, in the case of this project I do not believe that the minimum requirements are sufficient criteria for a detailed evaluation. This is chiefly because the project did not in fact achieve the minimum requirements (again, see below). It is therefore necessary to examine the reasons why this was the case, and evaluate whether these reasons could have been foreseen and/or avoided. To this end, during my evaluation of the project I shall keeping in mind the following criteria:

1. Was there sufficient understanding of the problem which the project aimed to address?
2. Could the problems which caused the project to fall short have been anticipated?
3. Could the above problems, once encountered, have been addressed more effectively?

6.2 Evaluation Against Minimum Requirements

The minimum requirements for this project are as follows:

1. A framework which facilitates the development of RoboCup agents by evolutionary computation.
2. An agent which can perform some individual behaviour(s) effectively and appropriately, having been generated by evolutionary techniques.

Quite clearly, the project has failed to meet these requirements. The framework has not been completed, and without the framework no agents could be evolved. I believe that the main reason for this is that the project's aims were not realistic in the timeframe, even though they seemed perfectly achievable back in September. I do not think that the difficulties which were found in attempting this project could have been anticipated; the problems which emerged were very deeply hidden within the project's structure, and did not reveal themselves until towards the very end of the project.

The problems which I faced are detailed in the rest of this chapter, alongside the steps which were taken to overcome them and an evaluation of whether or not they could have been avoided.

6.3 Project Scope

Given that the project ran out of time to complete the minimum requirements, it must immediately be asked whether the scope of the problem studied was too large for a final year project. I think the answer to this question is probably a cautious yes although, as mentioned above, this was pretty much impossible to know until well into the project. The problem itself was certainly clear enough, and the project's progression was smooth until the implementation phase. It was only once the project was well into the implementation phase that the complexity of creating the framework became obvious. I feel it is important to emphasise that the complexities which caused the problems during the implementation were not inherent to the problem of creating RoboCup agents by using genetic programming; they were rather problems caused by the idiosyncratic nature of the packages which I was trying to tie together. Both RoboCup and evolutionary computing are highly specialised applcation areas, and packages developed for use in these areas reflect this specialisation in the form of limited interoperability with other research areas. It is a simple fact that TriLearn was not created for use with evolutionary techniques, and also that BEAGLE was not designed for use with RoboCup; it was the disparities in their designs which caused the project to miss the minimum requirements.

This problem of disparities would, I feel, have been an issue regardless of the packages chosen. I reviewed my options in this regard before starting the implementation and I firmly believe that the packages I eventually used were the best choices available to me. The initial selection of the EO framework

was a mistake (see section 6.4.2), but I think that the selection of TriLearn and BEAGLE was justified. As explained in section 3.1, attempting to implement the whole structure from scratch *would* have been an obviously unrealistic project, so that was not an option, and even if I had chosen different packages to those which I eventually did, I would still inevitably have had problems trying to get them to work together, for exactly the reasons stated at the end of the previous paragraph.

6.4 Research

6.4.1 Understanding the Problem

Overall, I feel this area is the strongest part of the project. When the project was agreed at the start of the year, it was intended to be a research project conducting experiments with genetic programming. To this end, I carefully researched both RoboCup and GP until I felt that I understood the two domains well enough to be able to use them to conduct meaningful and useful experiments. I made sure that, at all stages, I was perfectly clear about what it was that I wanted to achieve. I think that the research, preparation and design sections of this report reflect this.

6.4.2 The EO Framework

The one area where I obviously slipped up in my preparation was in selecting the EO framework as my toolkit. Because of doing this, I lost nearly two weeks of implementation time; a fortnight which, with luck, might just have allowed me to finish the framework. The mistake I made here was in not doing detailed enough research into the different toolkits. EO and BEAGLE are both very large, complex products, and I failed to examine them closely enough. If I had perhaps attempted to implement a small GA or GP application under the two different systems, then the difference in usability between EO and BEAGLE might have been a bit more obvious; however even back in the research phase I was well aware that the project was going to be running on a tight schedule, and I did not believe that I had time to spare on a minute examination of the toolkits. I also have doubts about whether constructing a small system (which would have been the obvious way of assessing the relative merits of the two toolkits) would have actually swayed my decision; both toolkits had a very steep learning curve, and as I mentioned earlier in the report (section 5.1.1), I could have implemented a simple system with the EO framework just as easily as I could have done with BEAGLE. The conclusion that I have to draw from all this is that although the mistake in toolkit selection could have been avoided, it could not have been

easily avoided. I would have had to have done extremely in depth research to have had much chance of realising EO's limitations, and such research (assuming I were to look at all the toolkits in the same level of detail) would probably have lost me more time than I lost by having to switch toolkits part way through the project.

6.5 Project Management

As I mentioned in the sections above, I believe that the majority of the obstacles which saw this project stumble were not realistically avoidable. However, I think that in some cases I might have been able to respond to the problems better once they arose. For instance, when I encountered problems with the EO framework, I should probably not have persevered with it for quite so long. I think that after a few days looking at the framework I had spent enough time with it to realise that it would not be usable; however, I still persevered with it for a further week, in a futile attempt to understand it. That wasted week could have come in very useful at the end of the project.

I also persisted for too long in my original implementation plan, even after I started falling behind it. When the EO problem coupled with the complications in implementing the primitives meant that I was running significantly behind schedule, it might have been sensible to review the project plan and see whether some tasks could be removed or reordered in order to help reduce the deficit. One possible solution would have been to drastically reduce the number of primitives implemented in the first phase, move on to the next phase and then go back to finish the primitives if I had time. This would have given me more chance of completing the framework. Instead, I placed a bit too much faith in my ability to claw the time back simply through hard work, and ended up with a very nice, complete set of primitives and no GP framework to use them in.

I think this also points up a slight flaw in the implementation plan. The plan involved implementing the seperate parts of the system and then joining them together at the end, whereas with hindsight, it might have made things easier if I had first implemented a skeleton framework for communication between the different parts of the system, and then filled out the components after that. By the time this flaw became apparent, however, I was so far into the implementation that going back and undoing everything that I'd done was simply not an option.

As another general comment on my project management, I think that one of my failings during this project was that I didn't ask for help promptly enough when I needed it. I spent a lot of time

trying, often unsuccessfully, to figure out some problem on my own, when even if I was successful in solving the problem it took me longer than it would have done had I simply asked someone about it. If I had been more prepared to ask for assistance at various points during the project, I would probably have found the experience considerably less frustrating and been able to work out my problems more quickly. Also, I would probably have been able to find better solutions: there were often occasions during the implementation when I ended up solving one problem but creating another, and consultation with other more experienced developers would probably have lessened these cycles.

This critical analysis should not hide the fact that there were some positive aspects to my management of the project: my initial plans were sensible, and I experienced no significant problems until well into the implementation phase. Up until the issues with the EO framework and the unexpected complications during the implementation, the project was actually proceeding very smoothly, in large part because of my sticking to my plans and being sure to manage my time appropriately. It was only in the latter stages of the project that the management issues began to surface. The fact that the project failed to meet requirements has automatically set me looking for what went wrong in the project, but I believe that my responses to a lot of the problems I encountered were good. The sad truth is that the ones which I failed to handle well cost me severely in terms of the completion of the project, and so these are the ones which must be focussed on in this evaluation phase.

6.6 Final Status of the Framework

At the time of writing this report, the implementation of the framework is incomplete. However, I have good reason to believe that it could be completed using the techniques which I have outlined in section 5.3.1; see that section for my reasons for thinking this.

I am reluctant to estimate how long it would take to complete the framework, given the inaccuracy of most of my predictions on this score during the course of the project; however, I believe that, including time to allow for familiarisation with the code, it could be completed well within a month by a competent programmer. Having said that, some areas of the code are highly untidy, largely due to the large quantity of test methods that I've had to use for evaluating the performance of various different technologies, and these bits would need tidying up. The code is, as has been made clear throughout the report, complicated; I do not believe it would be a project that someone should get involved in if they are unsure of their programming abilities.

Chapter 7

Directions for Future Work

In the case of this project, the main direction for future work is obvious: complete the framework, and then complete the experiments which I had planned. I have talked about what would be required to complete the framework in section 6.6.

Once the framework were complete, it would open up a massive amount of possibilities for future work; not just the ability to conduct the experiments which I have outlined in this report, but also a wide variety of related ones. A complete framework would make it easy to conduct almost any experiments using genetic programming and RoboCup; the experimenter would be able to attempt the evolution of any RoboCup skill or strategy module that he or she cared to try. It is entirely plausible that the framework could, without too much effort, be used to evolve an entire team. This would prove the framework to be a highly useful tool in the RoboCup domain.

Of the many experiments which a fully-functioning framework would make possible, one direction which I would be particularly interested in seeing results from would be using self-adaptive mechanisms as part of the evolution (mentioned in section 2.3.2). It is possible to create an evolutionary algorithm which will change its own parameters over the course of the run. This is a relatively new technique but one which shows much promise. The idea can be applied to any of the EA's parameters: fitness function, mutation rate, population size etc ([6] provides a survey of the possibilities). It has been shown that these self-adaptive EAs can improve both the final result of the evolution and the speed

(measured in the number of generations) with which the optimum result is obtained (eg [4]), and I feel that self-adaptation is an idea which could be successfully applied to RoboCup in the future.

As mentioned in section 2.3.2, I particularly see a role for it in adapting the fitness function when evolving an entire team. As Wilson found, the randomly generated agents at the start of an evolutionary run are unlikely to be very good at even kicking the ball, let alone doing anything more complex such as scoring goals, and so it makes sense that the optimal fitness function during the early generations of a run would no longer be optimal towards the end of the run. I see this as being analogous to the way that a person who has never played football in his or her life before would be judged by different standards to those which would be applied to a professional footballer. Any form of self-adaptation would be interesting to try though; the fact that there is a rational argument for why adapting the fitness function might be successful does not diminish the possibilities offered by other forms of self-adaptation.

One restriction which affects the framework as it currently exists is its reliance on the basic dribbling, passing etc skills provided with the TriLearn release. Given that this release is now four years old, it is likely that more effective dribbling and passing skills have been developed, which would improve the performance of any team evolved using the system. It would be interesting to try replacing the standard TriLearn skills with more recently developed equivalents: it would provide a gauge of how far the RoboCup domain has progressed. In a similar vein, importing more complex skills into the framework for use as primitives (such as Brainstormers' multi-agent skills; see section 3.1.1.4) would allow for the evolution to operate at ever higher levels of abstraction. This would in turn mean that the evolutionary system's attention could be focussed on more complex strategic decision-making, rather than simply "what skill should I use next?", which could only improve the standard of RoboCup team which could be developed.

As the complexity of the RoboCup player agents increases, it seems sensible to expect that the complexity of the coaches will increase as well. As evolutionary techniques are proven to work well in highly complex scenarios, it might be interesting to attempt to create a functioning coach by using evolutionary techniques. The technical and theoretical challenges involved in this would be large (for instance it would probably be impossible to do what I have done in using the trainer to automate control of the evaluations), and implementing such a system would not benefit as I have from having reusable code readily available. However, it is an idea which could prove to be workable in the future, and which might prove to be a worthwhile addition to evolutionary computing's repertoire.

Bibliography

[1] D. Andre and A. Teller. Evolving Team Darwin United. In M. Asada and H. Kitano, editors, *RoboCup-98: Robot Soccer World Cup II*. Springer-Verlag, 1999.

[2] J. Aronsson. Genetic programming of multi-agent systems in the robocup domain. Master's thesis, Lund Institute of Technology, Lund University, 2003.

[3] M. Chen, E. Foroughi, F. Heinz, ZX. Huang, S. Kapetanakis, K. Kostiadis, J. Kummeneje, I. Noda, O. Obst, P. Riley, T. Steffens, Y. Wang, and X. Yin. *RoboCup Soccer Server User's Manual (for Soccer Server Version 7.07 and later)*, August 2002.

[4] V. Cicirello and S. Smith. Modeling GA performance for control parameter optimization. In D. Whitley, D. Goldberg, E. Cantu-Paz, L. Spector, I. Parmee, and H-G. Beyer, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2000)*, pages 235–242, Las Vegas, Nevada, USA, 2000. Morgan Kaufmann.

[5] N. Cramer. A representation for the adaptive generation of simple sequential programs. In J. Greffenstette, editor, *Proceedings of an International Conference on Genetic Algorithms and their Applications*. Morgan Kaufman, 1985.

[6] A. Eiben, R. Hinterding, and Z. Michalewicz. Parameter control in evolutionary algorithms. *IEEE Trans. on Evolutionary Computation*, 3(2):124–141, 1999.

[7] A. E. Eiben and J. E. Smith. *Introduction to Evolutionary Computing*. Natural Computing Series. Springer-Verlag, 2003.

[8] The RoboCup Federation. Robocup official website. `http://www.robocup.org/`. Last accessed on 23.04.2007.

[9] C. Gagné and M. Parizeau. Genericity in evolutionary computation software tools: Principles and case study. *International Journal on Artificial Intelligence Tools*, 15(2):173–194, April 2006.

[10] M. Keijzer, J. Merelo Guervós, G. Romero, and M. Shoenauer. Evolving objects: A general purpose evolutionary computation library. In *Selected Papers from the 5th European Conference on Artificial Evolution*, volume 2310 of *Lecture Notes In Computer Science*, 2001.

[11] J. Kok, N. Vlassis, and F. Groen. Uva trilearn 2003 team description. In D. Polani, B. Browning, A. Bonarini, and K. Yoshida, editors, *RoboCup 2003: Robot Soccer World Cup VII*. Springer-Verlag, 2004. Included on CD accompanying book.

[12] J. Koza. *Genetic Programming: On the Programming of Computers by Natural Selection*. MIT Press, 1992.

[13] J. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, 1994.

[14] J. Koza and R. Poli. A genetic programming tutorial. In E. Burke and G. Kendall, editors, *Search Methodologies: Introductory Tutorials in Optimization, Search and Decision Support*. Springer-Verlag, 2006.

[15] S. Luke. Evolving soccerbots: A retrospective. In *Proceedings of the 12th Annual Conference of the Japanese Society for Artifical Intelligence*, 1998.

[16] S. Luke. Genetic programming produced competitive soccer softbot teams for RoboCup97. In *Proceedings of the Third Annual Genetic Programming Conference (GP98)*, 1998.

[17] D. Montana. Strongly typed genetic programming. Technical Report 7866, Bolt Beranek and Newman Inc., 70 Fawcett Street, Cambridge, MA 02138, 1994.

[18] A. Di Pietro, L. While, and L. Barone. Learning in RoboCup keepaway using evolutionary algorithms. In *Proceedings of the Genetic and Evolutionary Computation Conference 2002*. Morgan Kaufmann, 2002.

[19] C. Reynolds. Competition, coevolution and the game of tag. In R. Brooks and P. Maes, editors, *Artificial Life IV: Proceedings of the Fourth International Workshop on the Synthesis and Simulation of Living Systems*. MIT Press, 1994.

[20] M. Riedmiller, T. Gabel, J. Knabe, and H. Strasdat. Brainstormers 2d - team description. In A. Bredenfeld, A. Jacoff, I. Noda, and Y. Takahashi, editors, *RoboCup 2005: Robot Soccer World Cup IX*. Springer-Verlag, 2006. Included on CD accompanying book.

[21] M. Riedmiller, T. Gabel, and H. Schultz. Brainstormers 2d public source code release 2005. Included with source code release; also at `http://www.ni.uos.de/fileadmin/user_upload/publications/riedmiller.gabel.schulz.bs05publicrelease.pdf`, 2005. website last accessed on 19/04/2007).

[22] P. Rogalinski and M. Piasecki. Team WROCLAW2002. `http://autonom.ict.pwr.wroc.pl/robocup_old/files/docs/we/wroclaw2002_long.pdf`. Team description for Team WROCLAW2002. Last accessed on 06.11.2006.

[23] P. Rogalinski and M. Piasecki. Wroclaw2002 team website. `http://autonom.ict.pwr.wroc.pl/robocup_old/english/english.html`. Last accessed on 12.11.2006.

[24] P. Stone and R. Sutton. Keepaway soccer: a machine learning testbed. In *RoboCup-2001: Robot Soccer World Cup V*. Springer-Verlag, Berlin, 2002.

[25] P. Stone, R. Sutton, and G. Kuhlmann. Reinforcement learning for RoboCup soccer keepaway. *Adaptive Behavior*, 13(9), 2005.

[26] P. Stone, R. Sutton, and S. Singh. Re-inforcement learning for 3 vs. 2 keepaway. In P. Stone, T. Balch, and G. Kraetszchmar, editors, *RoboCup-2002: Robot Soccer World Cup IV*. Springer-verlag, 2003.

[27] P. Stone, M. Veloso, and P. Riley. The Cmunited-99 champion simulator team. In M. Veloso, E. Pagello, and H. Kitano, editors, *RoboCup-99: Robot Soccer World Cup III*. Springer-Verlag, 2000.

[28] M. Taylor and P. Stone. Behavior transfer for value-function-based reinforcement learning. In *AAMAS '05: Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems*, pages 53–59, New York, NY, USA, 2005. ACM Press.

[29] BEAGLE Project Team. BEAGLE GP tutorial. `http://beagle.sourceforge.net/wiki/index.php/Documentation:Manual:Tutorial`. Website last accessed on 24/04/2007.

[30] BEAGLE Project Team. BEAGLE project website. `http://beagle.gel.ulaval.ca/index.html`. Last accessed on 17/04/2007.

[31] EO Project Team. EO interface documentation. Included in EO source code release; also at `http://eodev.sourceforge.net/eo/doc/html/index.html`. Website last accessed on 21/04/2007.

[32] EO Project Team. EO tutorial. Included in EO source code release; also at `http://eodev.sourceforge.net/eo/tutorial/html/eoTutorial.html`. Website last accessed on 21/04/2007.

[33] T. Walker, J. Shavlik, and R. Maclin. Relational reinforcement learning via sampling the space of first-order conjunctive feature. In P.Tadepalli, R. Givan, and K. Driessens, editors, *Proceedings of the Workshop on Relational Reinforcement Learning at ICML'04*, 2004.

[34] P. Wilson. Development of a team of soccer playing robots by genetic programming. Honours Thesis, Royal Melbourne Institute of Technology, Department of Computer Science, 1998.

Appendix A

Personal Reflection

One thing I can say with certainty is that I have learnt a great deal during the course of the project. This does not just apply to the more academic aspects of the project (RoboCup, genetic programming and C++ programming), but also to other areas. On the academic side, I am very pleased with the amount of knowledge I gained. I deliberately chose the areas of intelligent agents and evolutionary computing for my project because they were areas of strong appeal to me, and I found my research into these fields extremely interesting. Although I knew nothing about GP at the start of the project, I feel that I am now conversant in the field, as well as being far more clued up about the sorts of issues affecting research into artificial intelligence than I was previously.

Looking away from the academic knowledge, this was my first experience of trying to manage a large project. I have never learnt anything about project management strategies; my placement year gave me some insights into it, but nothing formal, and I think this is one area where I could have benefitted from having more knowledge at the start of the project. Some of my planning was not as detailed as it probably should have been, meaning that I often found myself moving onto the next phase of the project prematurely, and then having to go back to a previous stage and properly define what I was doing. This was far less efficient — not to mention far more stressful — than being able to progress through the project's stages in a smooth and controlled fashion.

When it came to the implementation phase, I think I had a tendency to get very caught up in one

small part of the implementation at a time, and this was detrimental to my view of the project as a whole. This is, I think, a natural tendency for me: I am very much at home looking at fine details, and I do have a certain perfectionist streak. However, these are traits which you can't afford to indulge too much in a final year project; there simply isn't sufficient time.

This is probably the one thing which I would be most insistent about in advising another project student: plan all stages of your project in excrutiating detail, and be sure to go through your plans with your supervisor to ensure that they are actually realistic. This is especially true when it comes to designing and implementing a software system: if you skimp on the planning, then at the very least you'll just have to go back and do it later, and at the worst you'll hit a huge problem which proper planning would have avoided and end up losing a lot of time. Also, the planning should allow plenty of time to write up. Writing 50 pages of good work (which could well actually be 70+ pages, taking appendices into account) takes a very long time; not just the initial writing up, but also the proof-reading and revising which needs to come afterwards, and which are so easily forgotten about. If you can, write up the early parts of the project, such as research and design, as you go along; this means you don't have the chance to forget it all before you write up, and also that you have plenty of time to get feedback on draft versions.

I would say that my slightly lackadaisical approach to project planning is the chief reason why I wasn't particularly happy with the implementation aspect of the project. Not delivering the minimum requirements was not just harmful in terms of potential marks but also highly unsatisfactory from a personal point of view: having to leave something unfinished is not something that I enjoy, especially something of which I started off with such high hopes. Ironically, these hopes were probably high because I found the subject matter so interesting; I genuinely wanted my project to result in me finding out something new and interesting. However, with hindsight this was never likely to happen in the compressed timespan of a final year project. There are good reasons why researchers do their jobs full time. With these initial ambitions, the lack of even a functioning framework at the end of the project was deeply disappointing.

In spite of the negative impression of my implementation, I think that the writing up process has enabled me to see the project in a far more balanced light. At the point where I decided that I really couldn't spare any more time for the implementation and absolutely *had* to focus on writing up, I was feeling extremely depressed and unhappy with the project. With writing up and going over the entire six months of the project though, I came to realise that although there were aspects of the project with

which I wasn't impressed, there were also areas which I can be pleased with. I am especially happy with the way I went about the research section of the project, and I also believe that while the implementation may not be complete, what I have done is at least done to the best of my ability. It wasn't a complete disaster, even if it felt like it at times. (I guess that's another pearl of wisdom to pass onto future students: things are rarely as bad as they seem).

I think it's fair to say that the mistakes I made in this project were largely related to management and organisation, rather than to a fundamental inability to work with the subject matter. These kinds of mistakes can be learnt from, and I certainly intend to do my utmost to avoid repeating them in the future.

Appendix B

TriLearn Skills

This appendix reproduces a section of the file `BasicPlayer.h` from the TriLearn source code release. `BasicPlayer.h` contains the definition of the class `BasicPlayer`, which is where TriLearn's pre-implemented agent skills are implemented. The section reproduced contains the method definitions for these skills.

```
(File headers, member variables and class definition removed)

  /////////////////////////// LOW-LEVEL SKILLS ////////////////////////////////

  SoccerCommand   alignNeckWithBody       (                                  );
  SoccerCommand   turnBodyToPoint         ( VecPosition   pos,
                                            int           iPos = 1           );
  SoccerCommand   turnBackToPoint         ( VecPosition   pos,
                                            int           iPos = 1           );
  SoccerCommand   turnNeckToPoint         ( VecPosition   pos,
                                            SoccerCommand com                );
  SoccerCommand   searchBall              (                                  );
  SoccerCommand   dashToPoint             ( VecPosition   pos,
                                            int           iCycles = 1        );
```

```
SoccerCommand   freezeBall              (                                   );
SoccerCommand   kickBallCloseToBody     ( AngDeg        ang,
  double        dKickRatio = 0.16  );
SoccerCommand   accelerateBallToVelocity( VecPosition   vel                 );
SoccerCommand   catchBall               (                                   );
SoccerCommand   communicate             ( char          *str                );
SoccerCommand   teleportToPos           ( VecPosition   pos                 );
SoccerCommand   listenTo                ( ObjectT       obj                 );
SoccerCommand   tackle                  (                                   );

////////////////////////// INTERMEDIATE SKILLS ////////////////////////////

SoccerCommand   turnBodyToObject        ( ObjectT       o                   );
SoccerCommand   turnNeckToObject        ( ObjectT       o,
                                          SoccerCommand com                 );
SoccerCommand   directTowards           ( VecPosition   posTo,
                                          AngDeg        angWhenToTurn,
                                          VecPosition   *pos = NULL,
                                          VecPosition   *vel = NULL,
                                          AngDeg        *angBody  = NULL    );
SoccerCommand   moveToPos               ( VecPosition   posTo,
                                          AngDeg        angWhenToTurn,
                                          double        dDistDashBack = 0.0,
                                          bool          bMoveBack = false,
                                          int           iCycles = 1         );
SoccerCommand   collideWithBall         (                                   );
SoccerCommand   interceptClose          (                                   );
SoccerCommand   interceptCloseGoalie    (                                   );
SoccerCommand   kickTo                  ( VecPosition   posTarget,
                                          double        dEndSpeed           );
SoccerCommand   turnWithBallTo          ( AngDeg        ang,
                                          AngDeg        angKickThr,
                                          double        dFreezeThr          );
SoccerCommand   moveToPosAlongLine      ( VecPosition   pos,
                                          AngDeg        ang,
                                          double        dDistThr,
                                          int           iSign,
                                          AngDeg        angThr,
```

```
                                        AngDeg        angCorr           );

  ////////////////////////// HIGH-LEVEL SKILLS ////////////////////////////////

  SoccerCommand   intercept               ( bool          isGoalie          );
  SoccerCommand   dribble                 ( AngDeg        ang,
                                            DribbleT      d                 );
  SoccerCommand   directPass              ( VecPosition   pos,
                                            PassT         passType          );
  SoccerCommand   leadingPass             ( ObjectT       o,
                                            double        dDist,
                                            DirectionT    dir = DIR_NORTH   );
  SoccerCommand   throughPass             ( ObjectT       o,
                                            VecPosition   posEnd,
                                            AngDeg        *angMax = NULL    );
  SoccerCommand   outplayOpponent         ( ObjectT       o,
                                            VecPosition   pos,
                                            VecPosition   *posTo = NULL     );
  SoccerCommand   clearBall               ( ClearBallT    type,
                                            SideT         s = SIDE_ILLEGAL,
                                            AngDeg        *angMax = NULL    );
  SoccerCommand   mark                    ( ObjectT       o,
                                            double        dDist,
                                            MarkT         mark              );
  SoccerCommand   defendGoalLine          ( double        dDist             );
  SoccerCommand   interceptScoringAttempt (                                 );
  SoccerCommand   holdBall                (                                 );

(Utility methods removed)
```

Appendix C

TriLearn Custom Data Types

This appendix briefly outlines the custom data types used by TriLearn which I needed to use in the project.

`SoccerCommand` An object representing a command or series of commands to be sent to the soccer server.

`VecPosition` An object representing a position, treated as a vector.

`AngDeg` Represents an angle in degrees; really only an alias (`typedef`) for the type `double`.

`ObjectT` An enumeration representing the different types of objects in the soccer server: the ball, the players, the goals etc.

Appendix D

BEAGLE Configuration File

This appendix reproduces the configuration used to control BEAGLE's operation in this project. The elements between the `<Register>` tags are parameters used by BEAGLE; the elements in the `<Evolver>` tags are the custom evolver (see section 5.1.2.3). The `"ec.term.maxgen"` element sets the number of generations that the evolution will run for; this is set to 1 for testing purposes only.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<Beagle>
  <Register>
    <Entry key="ec.term.maxgen">1</Entry>
  </Register>

  <Evolver>
    <BootStrapSet>
        <IfThenElseOp parameter="ms.restart.file" value="">
        <PositiveOpSet>
            <GP-InitHalfConstrainedOp/>
```

```
            <FYPEvalOp/>
            <GP-StatsCalcFitnessSimpleOp/>
        </PositiveOpSet>
        <NegativeOpSet>
            <MilestoneReadOp/>
        </NegativeOpSet>
        </IfThenElseOp>
        <TermMaxGenOp/>
        <MilestoneWriteOp/>
    </BootStrapSet>
    <MainLoopSet>
        <SelectTournamentOp/>
        <GP-CrossoverConstrainedOp/>
        <GP-MutationStandardConstrainedOp/>
        <GP-MutationShrinkConstrainedOp/>
        <GP-MutationSwapConstrainedOp/>
        <GP-MutationSwapSubtreeConstrainedOp/>
        <FYPEvalOp/>
        <GP-StatsCalcFitnessSimpleOp/>
        <TermMaxGenOp/>
        <MilestoneWriteOp/>
    </MainLoopSet>
    </Evolver>
</Beagle>
```