CHALMERS

Multi-Agent Systems

An Approach to Design and Implement Applicable Software Architecture, World Modeling, and Decision Making

Master of Science Thesis

ASADOLLAH NOROUZI

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, 2010

Multi-Agent Systems
An Approach to Design and Implement Applicable Software Architecture, World Modeling, and Decision Making

ASADOLLAH NOROUZI,

Examiner: Sven-Arne Andreasson

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden 2010

To my beloved wife, Fahimeh Masoumian

Abstract

The need to have well-structured and managed software architecture for control, behavior, and decision making of multi agent environments is inevitable. This is due to the fact that these intelligent agents are expected to perform within a dynamically complex environment where proper and quick response in a cooperative manner is a primary key to successfully complete a task. This thesis follows two main trends. The first trend is to design a model for the agents' software architecture, and the latter trend is to implement a nondeterministic decision making using electric fields and high-level decision making. Different layers are designed, defined, and implemented for the software architecture with focus on system adaptability, sustainability, and optimization. The approach proposed in this thesis for the agents' decision making is based on layered artificial intelligence implemented using vector-based fuzzy electric fields and a decision tree. Furthermore, an approach to model the world which is, in this thesis, called Agent Relative Polar Localization is introduced. This world model is based on fuzzy measurements and polar coordinates. This thesis also proposes a special supplementary flexible decision tree which may and can be modified based on the application under which it is intended to be used or implemented. The aim of designing such decision tree is to provide more flexibility for the decision making process. In order to optimize the overall performance of the system learning methods have been introduced to the system. The proposed system in this thesis has been implemented on soccer robots to evaluate the performance of the system and see that theoretical studies also can be applied to the real applications. The results show that the proposed system implemented on the soccer robots is reliable and robust.

Acknowledgements

In order to write this master thesis and to complete the underlying work, there are some people I want to give a special thank.

Many thanks to

- My supervisor, Prof. Changjiu Zhou, who continuously supported and supervised me throughout the work of this thesis. I am very thankful to him for leading me on the right direction and path of this thesis work.
- My father, Dr. Abdollah Norouzi, and my mother, Dr. Nayyereh Amini who have for years supported me through the challenges of my life and without whom I would not have been at this stage. I hope they would be proud of this work.
- Last but not least, my special words of appreciations and gratitude goes to my wife, Mrs. Fahimeh Masoumian, whose support gave me the right energy to initiate and finish this thesis work. I am indeed grateful to her for providing the best possible environment to work on this thesis.

Göteborg, 2010
Asadollah Norouzi

Contents

Chapter 1

Introduction

The development of multi-agent systems is one of the major topics of today research. It is a science that simultaneously involves many areas of today engineering. That is, various branches of computer science and engineering are involved such as software engineering and artificial intelligence. In many available research papers each of these branches are discussed separately, while it is the combination of both software engineering and artificial intelligence that lead to the development of an optimum and well performing multi-agent system.

A previously published outline for a theory of intelligence defines intelligence as "the ability to act appropriately in an uncertain environment, where appropriate action is that which increases the probability of success, and success is the achievement of behavioral goals." [1] That is, the intelligent system must be designed such that it increases the probability of success and reduces the probability of failure. Thus, the achievement of the aforementioned goal requires at least the following assets:

- A well-structured architecture design for the system software
- An efficient and reliable algorithm
- A well designed hardware (in case of an actual robot or autonomous machine involvement)

This thesis focuses on the first two assets. The basic idea for the proposed system architecture in this paper has been drawn from the layered biological architecture of human. Although the proposed design is not the same as is for human but is to some extend similar to it. The design is such that the agent is provided with a collection of analyzed data prepared by each individual layer which enable the agent to make decisions with higher probability of success. The layers are designed to be dependent of other layers and thereby become easily

upgradable without the constraint to make major changes to other layers. The issue of fault tolerance is also considered in the design to decrease the probability of system failure. However, the system is designed such that it would enter a recovery state in case of failure in the system.

The intelligence model proposed in this thesis is based on electric fields. Every object in the environment will be defined by an electric field. The electric fields of all objects trigger electric forces represented by vectors where each vector is consisted of two main characteristics: Direction and Magnitude. The agent makes decision based on the combination of the resultant vector of all electric forces vectors and the agent's current status in its decision tree. These vectors are also used for the agent's navigation through the environment. The electric fields resultant vector is also used for the agents' navigation path. The agent's decision tree is dependent on the application under which the agent will be performing. Therefore, its decision tree must be designed according to the tasks and goals that the agent must follow to complete its designated mission.

The major principle of the World Model proposed in this thesis is to replace the numerical measurements and introduce fuzzy definitions in the measurement system. That is, the volume of calculations is considerably reduced in the proposed system. This is, in fact, somewhat similar to the human measurement process where measurements are performed in a non-numerical way. The proposed world model in this thesis is called Agent Relative Polar Localization; ARPL in short. This model provides the advantage of not requiring a precise global positioning. That is, a relative polar localization is introduced to substitute the global positioning and consequently no complex algorithm is required for the agent localization process. This, thereby, reduces calculation errors and enhances the system performance.

1.1 Background

The work on this thesis takes place at the Advanced Robotics and Intelligent Control Center of the Singapore Polytechnic. This center was established in 1998 and is now one of the leading research centers in the Southeast Asia. This center aims to carry out research and development work in the related areas through collaborative work with prestigious local and overseas universities and institutions by embarking on consultancy projects and initiating innovative projects in Singapore.

This center provides good resources to carry out the work on multi-agent systems and provides the appropriate test beds for the system implementation and testing. The multi-cultural environment of this center is a good benefit for infor-

mation sharing and brainstorming.

The multi-agent environments are very complex and dynamic. Therefore, the system must be designed and implemented such that it would provide appropriate and quick response to the environment's variables in a way to accomplish the defined task for agents.

1.2 Objectives

The multi-agent systems are rapidly developing both in hardware aspects and in software aspects. Therefore, the system must be able to adapt itself with new upgrades easily and quickly with minimum costs for the adapting process. The system architecture plays a very important role here to achieve this objective[2].

The system reliability is a challenging issue when referring to multi-agent systems. The presence of errors malfunctions and crashes are inevitable due to the dynamics of such environments and continuous interaction of the agents with the surrounding environment. Thereby, the system must be able to recover from errors and crashes that may occur in a part of the system while avoiding other parts of the system to crash as a result of this. Again, the system architecture is the key for the accomplishment of this objective.

Finally, the system must be able to well perceive and understand its environment and make appropriate decisions to successfully accomplish the system's defined tasks and goals. Therefore, the decision making algorithms, world modeling, and other related areas must be well studied and implemented to meet the system objectives.

1.3 Limitations

Multi-agent system environment as has been mentioned before is very dynamic and complex. This thesis proposes both a system architecture and also decision making and world modeling algorithms for optimum performance of the agent within such environments. However, it must be noted that the proposed system, even though showing good results, cannot be considered as a final solution to the problem. This work proposes a good solution for the problem but requires further study and research to achieve better results.

1.4 Disposition

The rest of the report is organized as follows. Chapter 2 discusses the proposed architecture for the system. The system designed layers are introduced and described in this chapter. Chapter 2 also presents the details for generating the electric fields vectors. Chapter 3 discusses the proposed method for the world modeling called Agent Relative Polar Localization. This chapter provides details of how the world is being modeled in a relative way. It also compares the proposed method with other localization methods and shows the comparison results. Chapter 4 provides the studies done on the learning mechanisms of the system. It introduces the learning methods proposed in this thesis. Chapter 5 provides details on the implementation of the proposed system within the context of soccer robots. The details of the implemented layers are given in this chapter. Chapter 5 also discusses the details of proposed method to generate the decision tree. Chapter 6 concludes this thesis report.

Chapter 2

The System Architecture

The agents in multi-agent environments are either software or hardware. The main objective of the proposed architecture in this thesis is to design and develop a system to be compatible with both software and hardware agents. The advantage of having such architecture is the possibility of having one program that can run on both software and hardware agents without re-designing and re-developing the whole system. Obviously, the need to re-design and re-develop the whole system imposes extra costs and is also time consuming. It must be mentioned that having a fully program compatibility between software and hardware agents may not be possible in all applications, but the proposed architecture in this thesis requires minimum changes to some parts of the program in order to transfer from software agent to hardware and vice versa. Furthermore there are always updates and upgrades for various sections of both hardware and software; therefore, the system is designed to be adjustable with new software and/or hardware without making major changes and modifications to the program.

The proposed architecture is based on layers which are independent and are responsible for certain tasks. The layers communicate with one another using special communication protocols. In case of hardware and/or software upgrade, only the protocols can be modified to meet the new compatibility requirements and therefore the rest of the program can remain unchanged. The idea of this architecture is inspired from the OSI (Open System Interconnection) model. The OSI model was introduced in July 1979 which is a layered architecture comprising seven layers: Physical, Data Link, Network, Transport, Session, Presentation, and Application[3].

The proposed system architecture in this thesis comprises five collaborating layers (see figure 2.1) that have been designed to meet the aforementioned objectives. The layers are designed such that they remain independent while being in constant collaboration and communication to other layers in order to reach the

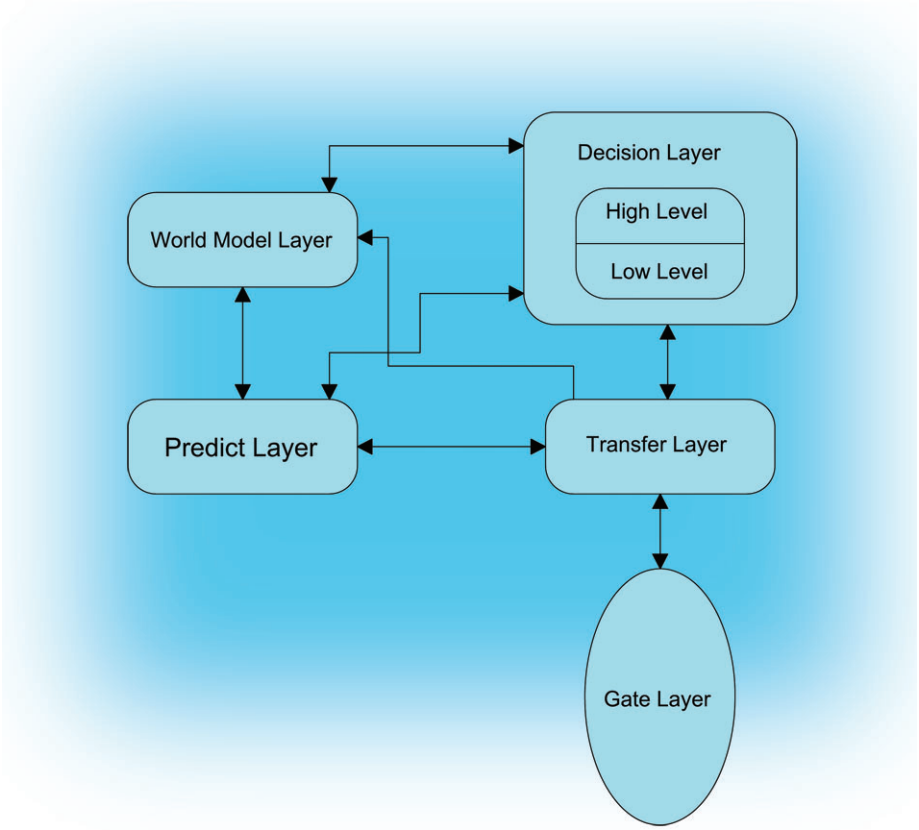

Figure 2.1: The illustration of the system architecture layers

overall goal of the system.

The layers are as follows:

1. Gate Layer
2. Transfer Layer
3. Decision Layer
4. World Model Layer
5. Predict Layer

The collaboration and communication between layers is through specific protocols that have been designed and implemented such that they would require min-

imal changes and modifications to become compatible with new software and/or hardware.

The main objective in the process of layers design and implementation is to provide independency and flexibility. This is of importance given the fact that multi-agent environments are environments of which undergo many changes and are considered very dynamic. Even though all layers are in connection with one another but the structure and detailed procedure of each layer is not an issue for other layers, in other words, the details have been abstracted and encapsulated. This design and implementation allows each layer to be updated or undergo structural changes without engaging changes and modifications to other layers. In these cases only the protocols need to be updated and modified.

The system fault tolerance has also been considered in the design and implementation. There is a special section in the system which is called Exception Handler. The Exception Handler is parallel process which monitors the performance of the system and detects any faulty in the system. It also listens and catches the operating system exception messages. The Exception Handler triggers appropriate action in case of a fault within the system and tries to recover the system. If the problem persists and system fails to recover the Exception Handler executes a restart command to force the system to reboot. In a case of reboot, if the memory of the system is not damaged to system will move to its last recorded state before the fault. This Exception Handler can also receive faulty reports from other layers. This feature is very useful especially when the layers communication is lost. The layers in some cases can detect malfunctions in other layers and consequently will report them directly to the Exception Handler.

2.1 Gate Layer

Gate Layer is the only layer that is in contact with the world and environment. All interactions between the agent and the environment go through this layer. In other words, it is a gateway between the agent and the surrounding world (environment). The Gate Layer has two principal tasks. The first task is to perceive and sense the environment through available input channels and/or sensors. These inputs include, but are not limited to, Vision, Ultrasonic Sensors, Proximity Sensors, Gyro, Encoders, etc. These input channels could either be hardware receiving information from an actual environment or be software channels receiving information from a simulated or virtual environment. The other task of the Gate Layer is to convert the commands and decisions made at the Decision Layer into executable and understandable commands for the output channels. The output channels include, but are not limited to, motors, actuators, etc. Similar to the input channels, the output channels could either be hardware or software.

In case of hardware or software upgrade for the input and output channels, only this layer needs to be modified and updated in order to be compatible with the new upgrades while the other layers remain unchanged.

Gate Layer only communicates with the Transfer Layer where there is a mutual communication between these two layers. The Gate Layer collects all the usable inputs and sends them to the Transfer Layer, and on the other hand the Transfer Layer sends the decision commands to the Gate Layer for execution.

2.2 Transfer Layer

Transfer Layer is mainly considered as a data parsing medium. This layer receives the input data from Gate Layer and prepares them to be sent to Decision, Predict, and World Model Layers. The Transfer Layer analyzes and parses the input data and converts them into the system's defined standard data types. There are inevitable errors and noises within the data received from the environment, therefore, this layer filters the noise and corrects the data where necessary. The Transfer Layer also receives the decision commands from the Decision Layer and converts them to understandable commands for the Gate Layer before sending them to that layer. To be more specific, the decision commands are sent to the Transfer Layer by the Low Level sub-layer of the Decision Layer. The Transfer Layer is one of the most significant and vital layers of the system because the most essential sections of the system which are world modeling and decision making use the data prepared by this layer.

This layer plays an important role in recognizing the faults within the system by analyzing the data that it receives from the Gate Layer and the Decision Layer. The data that are being transferred in the system are good assets of identifying probable errors in the system. These errors could either originate from the input channels when corrupted data is provided by these channels or from the output commands when corrupted commands are being issued.

In case of a software upgrade within the higher layers of the system, such as the Decision Layer, only the Transfer Layer requires modification to recognize the new upgrades.

2.3 Decision Layer

The Decision Layer is actually the main intelligent section of the system which is consisted of two sub-layers named Low-Level and High-Level. This division is due to the fact that not all the decisions necessarily lead to a physical action or

reaction whereas there are decisions for higher levels of intelligence[4]. Having said that, in this project the decisions leading to a physical action are called lo-level decisions and all other decisions are called high-level decisions.

The process of decision making at the Low-Level sub-layer is based on the inputs provided by the High-Level sub-layer and the World Model Layer. The types of the decisions made at the High-Level sub-layer are based on the application and environment under which the agent exists. An example type of a high-level decision that is shared by all multi-agent environments is the strategy of which the intelligent agents cooperatively conduct a task. The input from the World Model Layer is the resultant vector of all the generated electric field vectors. The Low-Level sub-layer combined both inputs and prepares a final decision ready for execution. This decision is then sent to the Transfer Layer and finally the Transfer Layer sends this command to the Gate Layer for execution.

The High-Level Decision recognizes and analyzes its surrounding environment using data generated by the World Model Layer and also takes the elements of the environment which can and may influence the actions of the agent into consideration and evaluation. The decision making process of the agent can be concisely broken down into the following:

- Make a High-Level decision using adequate information collected from the World Model Layer, the Predict Layer, and additional information or commands from other active elements of the environment.
- This decision made at the first step is shared with the electric field generator section of the World Model Layer to generate the corresponding electric fields vectors according to the High-Level decisions of the agent.
- The final part of the decision making process is done by the Low-Level sub-layer where the final decision is made using the combination of the results of the first above two steps.

2.4 World Model Layer

The World Model Layer is responsible to model the world based on the information that it receives from the Transfer Layer. The world model that is proposed in this thesis is such that every element of the world is modeled relative to the agent using the proposed Agent Relative Polar Localization method. The details of this method are further discussed later in the thesis. The other major task of the world Model Layer is to generate the electric fields vectors of all the world effective objects in the decision making process. The corresponding electric field vector for each object is generated using the hereunder equation:

$$F = k\frac{q_0 q_1}{r^2} \tag{2.1}$$

Where F is the electric force between the agent and an object in the environment, q_0 is the agent's electric charge which is set by the Decision Layer and dynamically varies to respond to the changing circumstances and conditions of the environment, q_1 is the electric charge of an object in the environment which is also set by the Decision Layer and just as the agent's charge is variable, r is the parametric distance between the agent and the object, k is the coulomb's charge law constant equivalent to $\frac{1}{4}\pi\varepsilon_0$. This value is set to one by default.

Generalizing the equation 2.1 to correspond to all the elements of the environment, we deduce the following equation for every object of the environment:

$$\vec{F_j} = k_i \sum_{i=1}^{n} (\frac{q_0 q_i}{r_i^2}, \theta_i) \tag{2.2}$$

Where Θ_i is the environment's objects angles relative to the agent, n is the number of objects of the environment, j is the vector type identity number, $\vec{F_j}$ is the resultant electric field vector for the jth vector type, k_i is a number that dynamically changes by the Decision Layer. This value is used to determine the *risk level* coefficient of agents. This need to have such coefficient was realized during the study and work of this thesis which can be used to let the agent become more flexible to the environment. The appropriate determination and use of this feature requires further studies. The vector types are defined depending on the application and environment of which the agent exists.

For example in case of soccer robots, as a result of equation 2.2 two vectors of $\vec{F_1}$ and $\vec{F_2}$ are calculated where the first one represents the move vector and the latter represents a possible kick vector (for either passing or shooting). Finally $\vec{F_T}$ is chosen from either of the calculated vectors using equation 2.3:

$$\vec{F_T} = \psi(\vec{F_1}, \vec{F_2}) \tag{2.3}$$

Where $\vec{F_T}$ is the resultant decision vector, ψ is a selective function that the Decision Layer uses to determine the optimum vector from the available and calculated input vectors.

2.5 Predict Layer

The Predict Layer is the forecasting side of information processing. The aim here is to derive information about how the surrounding world will be like at

some time $t_0 + \varepsilon_t$ in the future, for some $\varepsilon_t > 0$, by using data measured up to and including time ε_t. The predicted world is quiet useful for making High-Level decisions, especially in case of determining action strategies where prediction provides valuable sources for the decision making process. Therefore, the Predict Layer sends the predicted states to the Decision Layer for further processing.

This layer also plays an important role for error detection and correction. The presence of errors is inevitable and cannot be avoided, however must be controlled and reduced. The detection and correction of the errors and consequently providing the Decision Layer with more realistic information about the world, obviously, makes the decisions made by the Decision Layer more efficient. That is, this layer was designed to take control of this task. This layer receives information about the surrounding world from the Transfer Layer and will approximate the state of the world for n steps ahead. The number of n can be set and changed depending on the in the Predict Layer settings depending on the application of which the agent is running. If the difference between the newly received information from the Transfer Layer and the predicted information of the future world in the Predict Layer exceeds the difference factor threshold defined in the layer's settings, then Predict Layer sends a signal to the Exception Handler informing it that there might be a malfunction within one of the input channels. The Predict Layer replaces the invalid or incorrect data with an average value between the last valid information and the predicted state. This average is not necessarily a correct value but it would be close to the actual value in the world that allows the agent to make an appropriate decision in case of erroneous or unavailable input data. In this case a noise factor will also be calculated and produced by the Predict Layer which will be used for future calculations. This noise factor can be retrieved by the World Model Layer in case needed.

Chapter 3

Agent Relative Polar Localization

Agent Relative Polar Localization, ARPL in short, is a method for modeling agent's surrounding world based on polar coordinates of r and Θ where r represents distance and Θ represents angle. In this method, the agent perceives the location of the world's objects using polar coordinates relative to its location. Thereby, each object will have a distance and angle relative to the agent which produces a polar position vector. The collection of these polar position vectors will make the agent's world in a vector representation.

3.1 ARPL Exploration In A Case Study

A case scenario will be explored to better describe this method. In this scenario there are a number of agents as soccer players who are playing soccer in a soccer pitch (see figure 3.1). In this scenario the main input channel of the agent is a camera that is looking upwards to a hyperbolic mirror that provides omni-directional vision for the agent.

In this case there are two ways to express the values of distance between various objects in the environment (soccer pitch). The first would be the exact logarithmic value which is actually the logarithmic position of the object in the hyperbolic mirror. It should be noted that this position is not the exact metric value of the object distance since it is not recalculated using the hyperbolic equation of the mirror. The latter one which is used by the Decision Layer is the linguistic fuzzy representation of the distance. This is done by dividing the agent's circular visible range area into several logarithmic sections defined by linguistic quantities like "close", "near", "far", etc. In the example shown in figure 3.1 the position

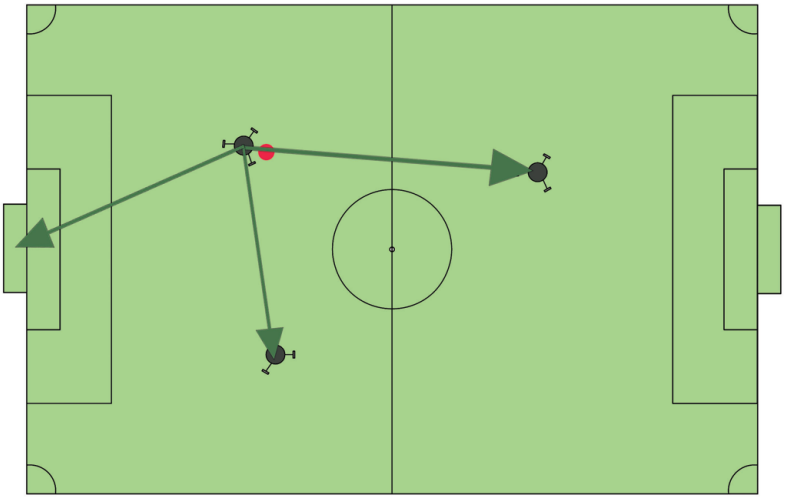

Figure 3.1: The agents localization using the proposed ARPL method.

vector of the ball would be as in equation 3.1, the position vector of the left goal would be as in equation 3.2, the position vector of the player towards the bottom of the field would be as in equation 3.3, and finally the vector position of the player towards the right goal would be as in equation 3.4.

$$r_{ball} = close, \quad \theta_{ball} = 348^\circ \tag{3.1}$$
$$r_{goal} = near, \quad \theta_{goal} = 210^\circ \tag{3.2}$$
$$r_{p1} = near, \quad \theta_{p1} = 280^\circ \tag{3.3}$$
$$r_{p2} = far, \quad \theta_{p2} = 350^\circ \tag{3.4}$$

Where r is the linguistic quantity of distance between the agent and object, Θ is the angle in degrees between the agent and object.

The magnitude of the linguistic distance ranges are increased exponentially from the closest point (tangent point) of the agent to the defined far most point. Agent Relative Polar Localization method eliminates the need of having static position references in the surrounding environment that allows the agent to avoid using Cartesian calculations (see equations 3.5 and 3.6). Instead, the agent can directly use the data provided by the vision which are distance and angle for every object.

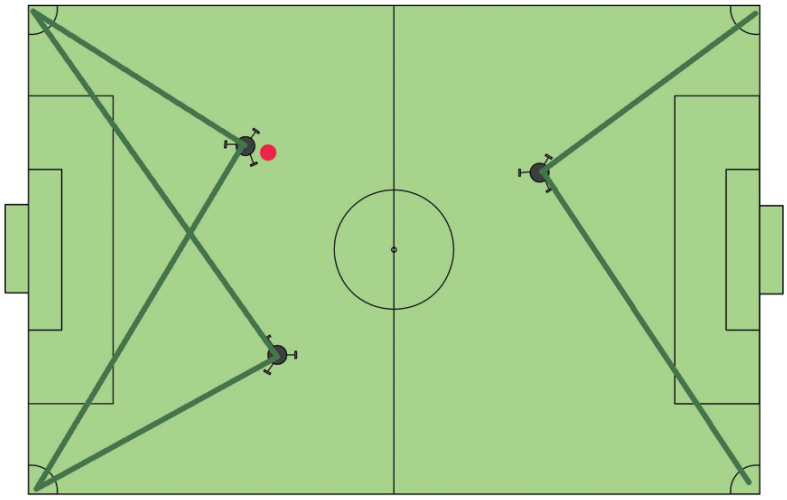

Figure 3.2: The agents localization using the method that requires two flag points to calculate the position.

$$x = r \cdot \cos(\theta) \tag{3.5}$$

$$y = r \cdot \sin(\theta) \tag{3.6}$$

Where x is the corresponding Cartesian *x-coordinate* of distance r and angle Θ, and y is the corresponding Cartesian *y-coordinate* of distance r and angle Θ.

3.2 Expriments and Results

The ultimate privilege of using polar coordinates over the Cartesian coordinates is the decrease of inevitable data calculations errors. Furthermore, the agent is free from modeling the global positions of the environment objects which reduces data processing and calculation. In other words, in this method the world modeling process does not depend on the stationary points (flags), and thereby the localization process become more reliable compared to other positioning and

localizing techniques where stationary points are used for localization and positioning (see figure 3.2 that depicts an example of localization using two flag points to calculate the position). One of the major drawbacks of such systems is that the failure of locating one of the stationary points would significantly produce localization errors. The Agent Relative Polar Localization method has been compared with three traditional positioning methods within the introduced scenario. The methods have been tested on three defined tasks for the agent to perform. The agent's first task is to track and follow a moving ball within the soccer pitch. The agent's second task is to move towards the goal while holding and dribbling the ball. The agent's last task is to move to an exact position within the soccer pitch. As can be seen in figure 3.3 the agent's performance is better for the first two tasks: "ball tracking" and "goto goal" while, on the other hand, the accuracy in "goto position" is only 60%. However, it should be noted that achieving high accuracy in "goto position" would not be possible even for a human soccer player without a GPS device! Thereby, it would be satisfactory if the agent successfully positions itself within a location close to the desired location.

Another experiment was carried out to test the performance of the agent's image and data processing in two localization methods. The two methods used in this experiment are this thesis proposed method and a traditional localization method using three points for positioning. Figure 3.4 depicts better performance of image processing and data fusion using Agent Relative Polar Localization compared to a traditional method of using three points for localization. The less number of vision frames per second using the traditional method is due to the fact that this and the similar methods require more calculations to acquire satisfactory results. The nature of the Agent Relative Polar Localization allows the agent to have fewer calculations of the data perceived from the environment and consequently being able to achieve a higher rate of image processing and also being able to process more usable data per processing time. The usable data here refers to the perceived data from the environment which can be used in the world modeling and decision making processes.

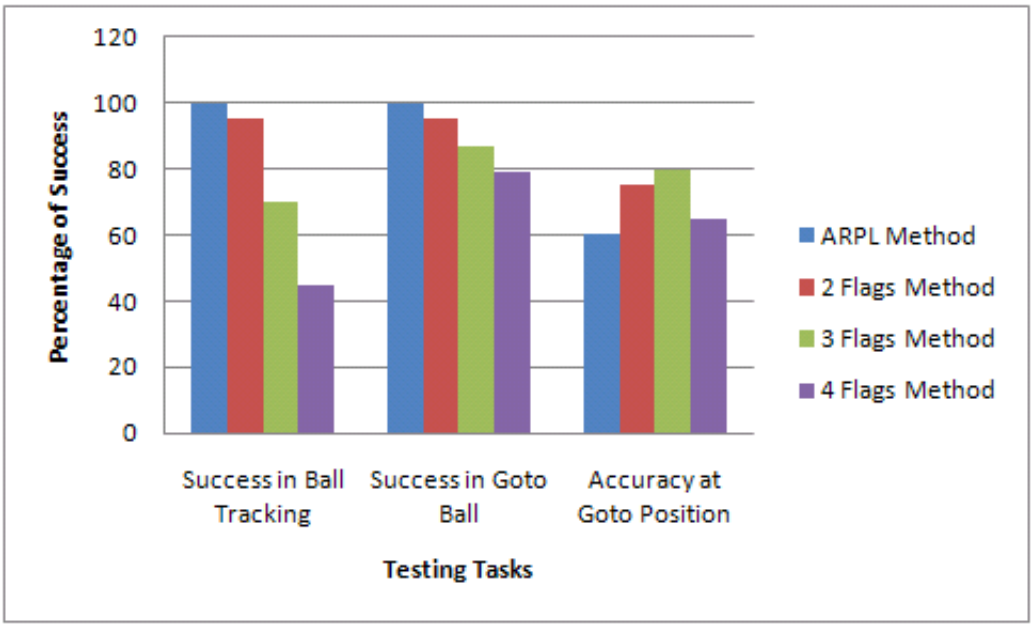

Figure 3.3: The performance comparison chart of ARPL localization method with three traditional localization methods where flag points are used.

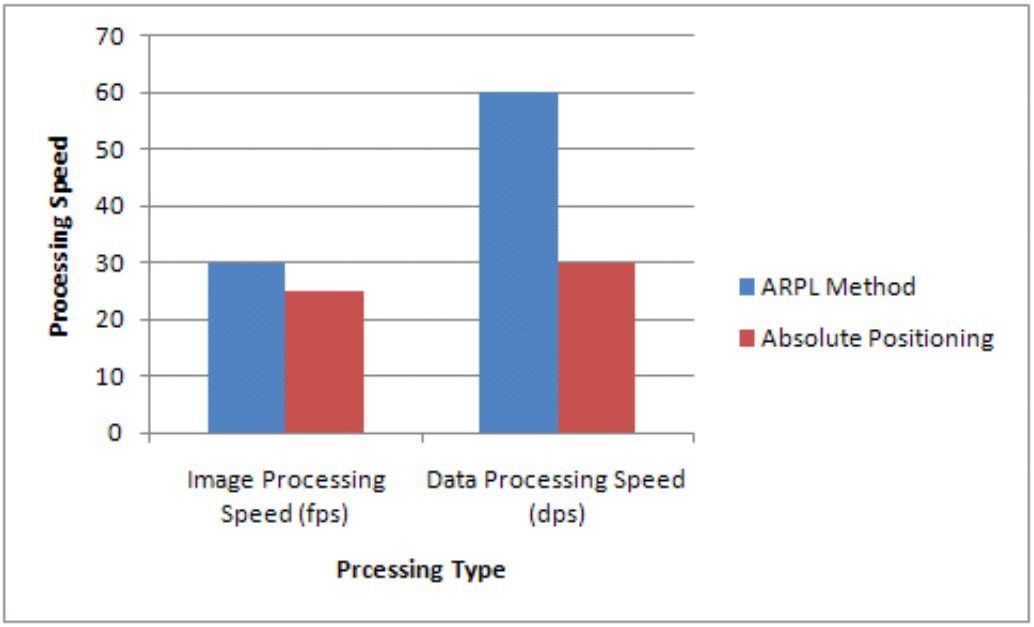

Figure 3.4: The data processing comparison chart of ARPL localization method with the traditional absolute positioning method that uses exact values for localization.

Chapter 4

Learning

Many researches on cooperative behavior suggest applying this behavior for a group of robots in a multi-agent system using learning methods, such as Q-Learning, Reinforcement Learning, Behavior Learning, etc. [5][6][7][8][9][10]. This thesis report focuses on the study of the Reinforcement learning and proposes two learning algorithms: Supervised Online Learning and Unsupervised Online Learning. In General, the Markov Decision Process (MDP)[11] is proposed for the learning section of this work. The MDP is a 4-tuple $(S, A, P.(.,.), R.(.,.))$, where S is a finite set of states, A is a finite set of actions, $P(s'|s,a)$ is a transition function, and $R(s,a)$ is the immediate reward received after the transition. Here, the agent takes an action ($a \in A$) at every state ($s \in S$). Then, the agent receives a reward $R(s,a)$ and reaches a new state s'. The new state s' is determined from the probability distribution $P(s'|s,a)$. The value $Q^*(s,a)$ of a given state-action pair *(s,a)* is determined by solving the Bellman equation[12]:

$$Q^*(s,a) = R(s,a) + \gamma \sum_{s'} P(s'|s,a) \underset{a'}{max} Q^*(s',a') \tag{4.1}$$

where γ is the discount rate and satisfies $0 < \gamma < 1$. The optimal value function Q^* can be found through value iteration by iterating over the Bellman equations until convergence[11]. Having had the state transition function P and the reward function R, the policy of which maximizes the expected discount reward can be calculated. Thereby, the optimal policy π is as follows[12]:

$$\pi(s) = argmax_a Q^*(s,a) \tag{4.2}$$

4.1 Supervised Online Learning

Figure 4.1 depicts a block diagram that illustrates the system's Supervised Online Learning. In conceptual terms, the supervisor or the coach is an intelligent agent that analyzes the performance of the performing agents outside the performance area. This online agent is able to interact with the agents and leading them to a better and effective performance having them learned from their performance. These learning processes are based on neural networks learning algorithms[13].

The form of supervised learning proposed in this thesis is the Error-Correction Learning. It is a closed-loop feedback system, but the unknown environment is not in the loop. As a performance measure for the system we may think in terms of the mean-square error or the sum of squared errors over the training sample, defined as a function of the free parameters of the system. This function may be visualized as a multidimensional error-performance surface or simply error surface, with the free parameters as coordinates. The true error surface is averaged over all possible input-output examples. Any given operation of the system under the teacher's supervision is represented as a point on the error surface. For the system to improve performance over time and therefore learn from the teacher, the operating point has to move down successively toward a minimum point of the error surface; the minimum point may be a local minimum or a global minimum. Our supervised learning system is able to do this with the useful information it has about the gradient of the error surface corresponding to the current behavior of the system. The gradient of an error surface at any point is a vector that points in the direction of steepest descent. In fact, in the supervised learning from examples, the system may use an instantaneous estimate of the gradient vector, with the example indices presumed to be those of time. The use of such an estimate results in a motion of the operating point on the error surface that is typically in the form of a random walk. Nevertheless, the proposed algorithm is designed to minimize the cost function. Thus, having this algorithm, using an adequate set of input-output examples, and enough time permitted to do the training, enables the supervised learning system to perform such tasks as pattern classification and function approximation.

4.2 Unsupervised Online Learning

The proposed system's Online Learning comes into action within the performing field of system. The Online Learning or learning without a teacher plays an important role enabling the system to be adaptive with environment variables[13]. Online Learning enables the agent that is using this system to increase its level of intelligence learning and experiencing from its action environment. In Online

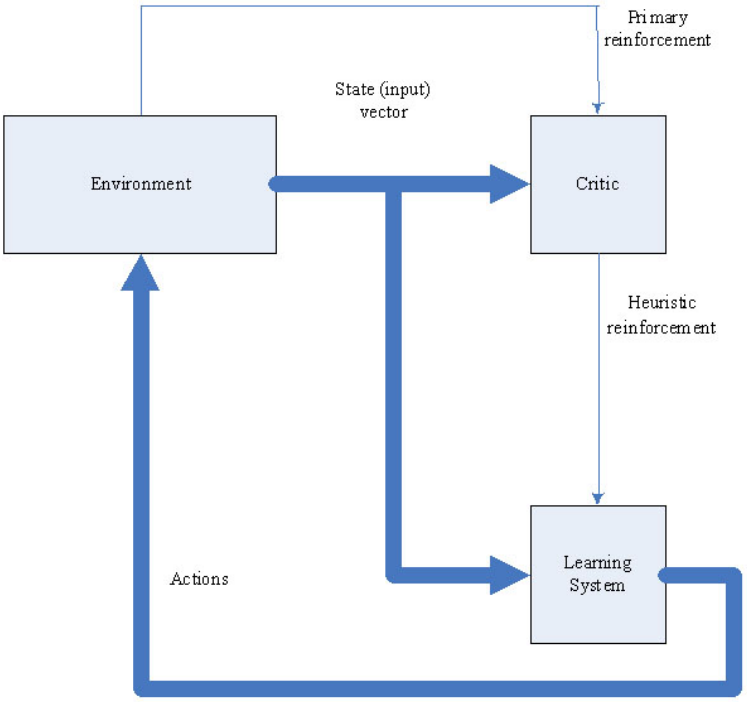

Figure 4.1: The block diagram of the proposed Supervised Online Learning.

Learning or learning without a teacher, as the name implies, there is no teacher to oversee the learning process. That is to say, there are no labeled examples of the function to be learned by the network. Here the reinforcement learning is used. In reinforcement learning, the learning of an input-output mapping is performed through continued interaction with the environment in order to minimize a scalar index of performance. Figure 4.2 shows the block diagram of the proposed reinforcement learning system built around a critic that converts a primary reinforcement signal received from the environment into a higher quality reinforcement signal called the heuristic reinforcement signal, both of which are scalar inputs. This system is designed to learn under delayed reinforcement, which means that the system observes a temporal sequence of stimuli (i.e. state vectors) also received from the environment, which eventually result in the generation of the heuristic reinforcement signal. Notwithstanding the difficulties of the delayed-reinforcement learning, this learning is very appealing. It provides the basis for the system to interact with its environment, thereby developing the ability to learn to perform a prescribed task solely on the basis of the outcomes of its experience that result from the interaction. In fact, the Decision Layer and the learning process are in contact with one another so that the learning process will be aware of the non-successful decisions made by Decision Layer.

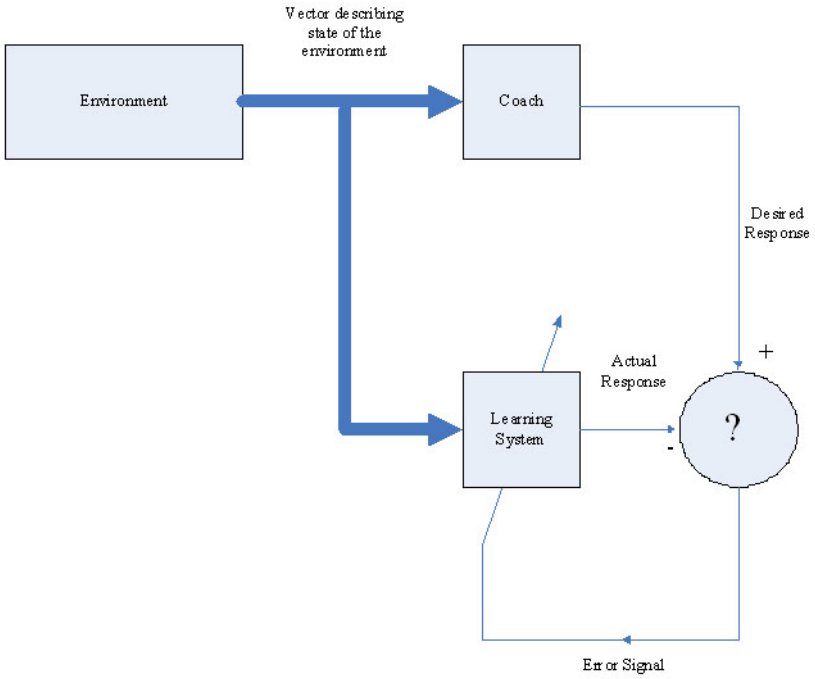

Figure 4.2: The block diagram of the proposed Unsupervised Online Learning.

Chapter 5

The System Implementation for Soccer Robots

The robotic soccer is known to be an appropriate and applicable environment for cooperative behavior where various multi-agent problems can be studied. It can be considered as a standard problem of multi-agent system and cooperative computation. Soccer robot international games, like RoboCup, are very useful testbeds to investigate and analyze various approaches to coordinate multi-agent systems operating in non-deterministic real environments[6][14][8]. Therefore, this system is implemented for F-2000 robots of the RoboCup MiddleSize League.

The program is implemented by C++ programming language under the Linux operating system. The program was installed for robots of MiddleSize League participating in RoboCup 2010 competitions that was held in Singapore during 19-25 June. The MiddleSize League is the soccer match between two teams of each 5 autonomous robots playing in a 18mx12m soccer pitch. The moment the match referee starts the match robots start to play soccer without any human interference.

The program comprises many sections, but this report focuses on the principal sections of the program that have been discussed in the thesis. The core part of the program is the design and implementation of the layers that have been discussed in chapter 2.

5.1 The Gate Layer Implementation

Gate Layer as discussed before is responsible to directly interact with real physical environment. This layer is defined as a namespace that includes three subnamespaces: Communication, MotorControl, and VisionTunnel.

The Communication namespace contains three classes: CAN, Network, and SHM. The CAN class is responsible to provide communication with the CAN bus communication port with the motor drivers (see program 5.1 for the class declaration).

```
#ifndef ClassCAN_H
#define ClassCAN_H

#include <libpcan.h>
#include "pgstream.h"
#include "pglib.h"

using namespace PG::std;

namespace PG{
  namespace SHIAI{
    namespace GateLayer{
      namespace Communication{
        namespace CAN{
          class ClassCAN{
            public:
              ClassCAN(/*add configuration into*/);
              ~ClassCAN();
              bool  Connect();
              bool  Disconnect();
              bool  SendSDO(int COBID,int len=1, int data0=0x00,
                      int data1=0x00,int data2=0x00,int data3=0x00,
                      int data4=0x00,int data5=0x00,int data6=0x00,
                      int data7=0x00);
              void  print_diag();
              int   GetReadNumber();
              bool  GetMessage(short* buff, short opcdolsb, short opcodemsb);
            private:
              HANDLE  handle;
              __u16   BTR0BTR1;
              char*   DevNode;
              bool    bConnectByDev;
              TPCANMsg Msg;
              bool    ShowDebugInfo;
              bool    CheckErr(DWORD err, char* sender);
          };
        }
      }
    }
  }
}

#endif
```

Program 5.1: CAN class declaration

The Network class is responsible for the providing communication via wireless and LAN with other agents and objects existing in the environment (see program 5.2 for the class declaration).

```
#ifndef PG_GATELAYER_COMMUNICATION_NETWORK_UDPClassUDPBase_H
#define PG_GATELAYER_COMMUNICATION_NETWORK_UDPClassUDPBase_H

#include <glib.h>
#include <gnet.h>
#include <iostream>

using namespace std;
//#define PGOut cout
//#define PGError cout

namespace PG{
  namespace SHIAI{
    namespace GateLayer
      {
      namespace Communication
        {
        namespace Network
          {
          namespace UDP
            {
              class ClassAddress
              {
                public:
                GInetAddr* Address;
                ClassAddress()
                {
                  Address = NULL;
                }
                bool operator == (ClassAddress param)
                {
                  return gnet_inetaddr_equal(Address, param.Address);
                }
                bool operator != (ClassAddress param)
                {
                  return  !gnet_inetaddr_equal(Address, param.Address);
                }
                void operator = (ClassAddress param)
                {
                  if (Address)
                    gnet_inetaddr_delete (Address);
                  Address = gnet_inetaddr_clone(param.Address);
                }
                void operator = (GInetAddr* addr)
                {
                  if (Address)
                    gnet_inetaddr_delete (Address);
```

```
            Address = gnet_inetaddr_clone(addr);
          }
        };
        typedef   void (*UDPEvent)(void*, int, ClassAddress);
        namespace UDPConnection
        {
          enum UDPConnections
          {
            Server,
            Client,
            NoInited
          };
          typedef UDPConnections Type;
        }
        struct StructUDPDatagram
        {
          bool                  Broadcast;
          char                  Data[800];
          int                   Lenght;
        };
        class ClassUDPBase
          {
          public:
            ClassUDPBase();
            ~ClassUDPBase();
            bool  InitSever(int port, UDPEvent EventManager);
            bool  InitClient(int port, char* host, UDPEvent EventManager);
            bool  Send(char* addr, void* data, int datalenght);
            bool  Send(int clientnumber, void* data, int datalenght);
            bool  Send(void* data, int datalenght);
            void*  Get();
            bool  Send(ClassAddress addr, void* data, int datalenght);
            void Close();
            static UDPEvent          EventFunc;
          private:
            UDPConnection::Type    ConnectionType;
            GInetAddr*             Addr;
          };
      }
    }
  }
}

#endif
```

Program 5.2: Network class declaration

The SHM class creates a shared memory for all the input/output data being transferred in or out of the Gate Layer (see program 5.3 for the class declaration).

```
#ifndef PG_GATELAYER_COMMUNICATION_SHMSHAREDMEMORY_H
#define PG_GATELAYER_COMMUNICATION_SHMSHAREDMEMORY_H

#include <sys/shm.h>

namespace PG{
  namespace SHIAI{
    namespace GateLayer{
      namespace Communication
        {
        namespace SHM
          {
          namespace SHMConnections
          {
            enum SHMConnections
            {
              Master,
              Slave
            };
            typedef SHMConnections Type;
          }
          namespace SHMStatus
          {
            enum SHMStatuses
            {
              Reading          = 'R',
              Writing          = 'W',
              DataReady        = 'D',
              Ready2Send         = 'S',
              Ready          = 'X',
              Open           = 'O'
            };
            typedef SHMStatuses Type;
          }
          class ClassSharedMemory
            {
            public:
              ClassSharedMemory();
              ~ClassSharedMemory();
              bool Open(int shmkey, int Masterbuffersize, int Slavebuffersize,
                    SHMConnections::Type shmtype);
              bool Send(void* data, int datalength);
              void*  Get();
              bool SetReady();
            private:
              SHMConnections::Type ConnectionType;
              char*    MasterBuffer;
              char*    SlaveBuffer;
              int      MasterBufferSize;
              int      SlaveBufferSize;
```

```
            char*    Buffer[1000];
            bool     SharedMemoryCreated;
          };
      }
    }
  }
}

#endif
```

Program 5.3: SHM class declaration

The MotorControl namespace contains one class which is responsible to send commands for the robot's motors. It also captures the feedback from the motor drivers to monitor the performance of the robot motors (see program 5.4 for the class declaration).

```
#ifndef PG_GATELAYER_MOTORCONTROLMOTORCONTROL_H
#define PG_GATELAYER_MOTORCONTROLMOTORCONTROL_H

#include "can.h"

using namespace PG::std;
using namespace PG::SHIAI::GateLayer::Communication::CAN;

namespace PG{
  namespace SHIAI{
    namespace GateLayer{
      namespace MotorControl
        {
        namespace MotorSet
        {
          enum MotorSetType
          {
            A   = 10,
            B   = 11,
            C   = 12,
            AB  = 20,
            AC  = 21,
            BC  = 22,
            ABC = 30,
          };
          typedef MotorSetType Type;
        }
        class ClassMotorControl
        {
          public:
          ClassMotorControl();
          ~ClassMotorControl();
          bool Init();
```

```
        bool SetAccelerationProfile4Motor(MotorSet::Type motor, long vel_A=0,
                        long vel_B=0, long vel_c=0);
        bool SetDecelerationProfile4Motor(MotorSet::Type motor, long vel_A=0,
                        long vel_B=0, long vel_c=0);
        bool SetEmergencyDecelerationProfile4Motor(MotorSet::Type motor, long vel_A=0,
                            long vel_B=0, long vel_c=0);
        bool SetMotionProfileType(int type=0);
        bool DisableMotor(MotorSet::Type motor);
        bool EnableMotor(MotorSet::Type motor);
        bool StopMotor(MotorSet::Type motor);
        bool SetSpeed4Motor(MotorSet::Type motor,long vel_A=0, long vel_B=0,
                    long vel_C=0);
        bool IsDriverEnabled();
    private:
        int  DriverID_A;
        int  DriverID_B;
        int  DriverID_C;
        long Acceleration_A;
        long Acceleration_B;
        long Acceleration_C;
        long Deceleration_A;
        long Deceleration_B;
        long Deceleration_C;
        long EmergencyDec_A;
        long EmergencyDec_B;
        long EmergencyDec_C;
        long MaxProfileVelocity;
        int  ProgressBarID;
        int  MaxTriesToSendCommand;
        int  MaxTriesToReset;
        int  InterInitSleep;
        int  AfterCommandSleep;
        int  BeforeReadSleep;
        bool ShowDebugInfo;
        ClassCAN CAN;
        bool FirstRun;
        bool Reset();
        bool SetoperationMode();
        bool SetMaxProfileVelocity(long);
        void ShowVelocities(char* str, MotorSet::Type motor, long vel_A=0,
                    long vel_B=0,long vel_C=0);
        bool IsDriverSwitchedOn();
        bool IsDriverDisabled();
        bool GetDriverStatus(int opcode, short& stat_A, short& Stat_B, short& Stat_C);
        inline short FirstByte(long num)
        {
          return num & 0x000000ff;
        }
        inline short SecondByte(long num)
        {
          return (num >>8) & 0x000000ff;
```

```
            }
            inline short ThirdByte(long num)
            {
              return (num>>16) & 0x000000ff;
            }
            inline short FourthByte(long num)
            {
                return (num>>24) & 0x000000ff;
            }
            inline char* MCT2String(MotorSet::Type motor)
            {
              switch (motor)
              {
                case MotorSet::A:
                  return "Motor A";
                case MotorSet::B:
                  return "Motor B";
                case MotorSet::C:
                  return "Motor C";
                case MotorSet::AB:
                  return "Motors A & B";
                case MotorSet::AC:
                  return "Motors A & C";
                case MotorSet::BC:
                  return "Motrs B & C";
                case MotorSet::ABC:
                  return "Motors A, B & C";
              }
              return NULL;
            }
          };
        }
      }
    }
}

#endif
```

Program 5.4: MotorControl class declaration

The VisionTunnel namespace contains one class which is responsible to capture the image from the camera that is used as the principal vision source of the robot. It collects all the useful data and stores them in a *struct* (see program 5.5 for the class declaration).

```
#ifndef PG_GATELAYER_VISIONTUNNELVISIONTUNNEL_H
#define PG_GATELAYER_VISIONTUNNELVISIONTUNNEL_H

#include "sharedmemory.h"
#include "omniwheel.h"
```

```
namespace PG{
  namespace SHIAI{
    namespace GateLayer
    {
      namespace VisionTunnel
      {
        namespace VisionObject
        {
          enum VisionObjectIDType
          {
            Ball              = 100,

            TmPlayer          = 10,
            OpPlayer          = 20,

            TmGoalRight       = 12,
            TmGoalLeft        = 13,

            OpGoalRight       = 22,
            OpGoalLeft        = 23,

            TmOut             = 14,
            OpOut             = 24,

            LeftOut           = 15,
            RightOut          = 25,

            MidLine           = 50,

            TmRightCorner     = 16,
            TmLeftCorner      = 17,

            OpRightCorner     = 26,
            OpLeftCorner      = 27,

            TmLeftFlagPost    = 18,
            TmCenterFlagPost  = 19,
            TmRightFlagPost   = 30,

            Center            = 51,

            OpLeftFlagPost    = 28,
            OpCenterFlagPost  = 29,
            OpRightFlagPost   = 40,

            CoachChair        = 80,

            Obstacle          = 90,

            ReturnMarker      = 150,
```

```
      HalfTimeColor       = 160,

      Unknown             = 200
    };
    typedef VisionObjectIDType Type;
  }
  namespace PlayerCommands
  {
    enum PlayerCommandType
    {
      HalfTimeColors,
      CalibrationProfile
    };
    typedef PlayerCommandType Type;
  }
  namespace GoalColors
  {
    enum GoalColorsType
    {
      Yellow,
      Blue
    };
    typedef GoalColorsType Type;
  }
  namespace TeamColors
  {
    enum TeamColorType
    {
      Magenta,
      Cyan
    };
    typedef TeamColorType Type;
  }
  namespace ProgramID
  {
    enum  ProgramIDType
    {
      Player,
      Vision
    };
    typedef ProgramIDType Type;
  }
  struct StructVisionObject
  {
    VisionObject::Type      ID;
    double                  Distance;
    double                  Angle;
  };
  struct StructHalfTimeColors
  {
```

```
        GoalColors::Type   GoalColor;
        TeamColors::Type     TmColor;
      };
      struct StructPlayerCommand
      {
        PlayerCommands::Type   Type;
        char                   Data[200];
        unsigned short         DataLength;
      };
      class ClassVisionTunnel
      {
        public:
          ClassVisionTunnel();
          ~ClassVisionTunnel();
          StructPlayerCommand* GetPlayerCommand();
          bool Send2Player(StructVisionObject* viewedworld, int numofobjects);
          StructVisionObject* GetViewedWorld();
          bool SendCommand(StructPlayerCommand command);
          bool Init(ProgramID::Type pid );
        private:
          PG::SHIAI::GateLayer::Communication::SHM::ClassSharedMemory SHM;
          bool TunnelInitialized;
          bool TunnelCreated;
      };
    }
  }
}

#endif
```

Program 5.5: VisionTunnel class declaration

5.2 The Transfer Layer Implementation

Transfer Layer is the data processor, analyzer, and converter part of the program. This layer is defined as a namespace containing three namespaces: Intranet, Robot, and WorldMaker.

The Intranet namespace comprise one class to manage the communication within the different sections of the robot. This part is also considered as to be the Intranet portal of the robot (see program 5.6 for the class declaration).

```
#ifndef PG_TRANSFERLAYER_IntranetIntranet_H
#define PG_TRANSFERLAYER_IntranetIntranet_H

#include "udpbase.h"
#include "MSL-commands.h"
```

```
namespace PG{
  namespace SHIAI{
    namespace TransferLayer{
      namespace Intranet
      {
        namespace PacketType
        {
          enum PacketTypes
          {
            Coach,
            Player
          };
          typedef PacketTypes Type;
        }
        typedef   void (*IntranetEvent)(void*, PacketType::Type);
        namespace AgentType
        {
          enum AgentTypes
          {
            Coach,
            Player,
            Goalie
          };
          typedef AgentTypes Type;
        }
        namespace PlayerDataType
        {
          enum PlayerDataTypes
          {
            World,
            BallOwning,
            GUI,
            Report,
            BugReport,
            DecisionVector,
            VisionColors
          };
          typedef PlayerDataTypes Type;
        }
        namespace CoachDataType
        {
          enum CoachDataTypes
          {
            World,
            BravityCoeff,
            VirtualPoint,
            Strategy,
            Command,
            RefereeCommand,
            VisionCalibrationData,
```

```
      HalfTimeColor,
      ManualControlCommand
    };
    typedef CoachDataTypes Type;
  }
  namespace RefereeCommand
  {
    enum RefereeCommands
    {
      Stop    = COMM_STOP,
      Start = COMM_START,
      Halt   = COMM_HALT ,
      Ready = COMM_READY,
          // game flow commands
      FirstHalf = COMM_FIRST_HALF,
      HalfTime = COMM_HALF_TIME,
      SecondHalf= COMM_SECOND_HALF,
      EndGame= COMM_END_GAME,
      Cancel = COMM_CANCEL,
        // goal status
      GoalMagenta= COMM_GOAL_MAGENTA,
      GoalCyan= COMM_GOAL_CYAN,
      SubGoalMagenta = COMM_SUBGOAL_MAGENTA,
      SubGoalCyan = COMM_SUBGOAL_CYAN,
      Restart = COMM_RESTART,
      KickOffMagenta= COMM_KICKOFF_MAGENTA,
      KickOffCyan= COMM_KICKOFF_CYAN,
      FreeKickMagenta= COMM_FREEKICK_MAGENTA,
      FreeKickCyan= COMM_FREEKICK_CYAN,
      GoalKickMagenta= COMM_GOALKICK_MAGENTA,
      GoalKickCyan= COMM_GOALKICK_CYAN,
      ThrowinMagenta= COMM_THROWIN_MAGENTA,
      ThrowinCyan= COMM_THROWIN_CYAN,
      CornerMagenta= COMM_CORNER_MAGENTA,
      CornerCyan= COMM_CORNER_CYAN,
      PenaltyMagenta= COMM_PENALTY_MAGENTA,
      PenaltyCyan= COMM_PENALTY_CYAN,
      DroppedBall= COMM_DROPPED_BALL,
      Dummy= COMM_DUMMY
    };
    typedef RefereeCommands Type;
  }
  struct StructPlayerDatagram
  {
    int             PlayerNumber;
    PlayerDataType::Type  Type;
    void*                 Data;
  };
  struct StructCoachDatagram
  {
    CoachDataType::Type   Type;
```

```
        void*                   Data;
      };
      class ClassIntranet
      {
        public:
          ClassIntranet();

          ~ClassIntranet();
          static void evmUDPServer(void* data, int datalength, PG::SHIAI::GateLayer::
                      Communication::Network::UDP::ClassAddress addr);
          static void evmUDPClient(void* data, int datalength, PG::SHIAI::GateLayer::
                      Communication::Network::UDP::ClassAddress addr);
          bool Init(AgentType::Type agent, IntranetEvent evm, int port,
                char* host = NULL);
          bool Broadcast(void* data, int datalenght, int headerlength);
          bool Send(int playernumber, void* data, int datalenght, int headerlength);
          bool Send(void* data, int datalenght, int headerlength);
          static IntranetEvent  EventManager;
          static PG::SHIAI::GateLayer::Communication::Network::UDP::
              ClassAddress  Client[6];
          static PG::SHIAI::GateLayer::Communication::Network::UDP:: ClassUDPBase UDP;
          static bool ShowDebugInfo;
        private:
          AgentType::Type Agent;
          char buff[250];
      };
    }
  }
}

#endif
```

Program 5.6: Intranet class declaration

The Robot namespace comprise three classes: Monitor, OmniVision, OmniWheel. The Monitor class is responsible to monitor the performance of the system and receive exception signals and error reports. It analyzes the error and malfunction type, and then triggers a recovery solution for the system (see program 5.7 for the class declaration).

```
#ifndef PG_SHIAI_TRANSFERLAYER_ROBOT_MONITORMONITOR_H
#define PG_SHIAI_TRANSFERLAYER_ROBOT_MONITORMONITOR_H

namespace PG
  {
  namespace SHIAI
    {
    namespace TransferLayer
      {
      namespace Robot
```

```
        {
        namespace Monitor
          {
          class ClassMonitor
            {
            public:
              ClassMonitor();
              ~ClassMonitor();
        bool Error;
        bool Recovered;
        int RecStatus;
        private:
          bool startMonitor();
          bool endMonitor();
          bool getError();
          bool sendError();
          bool Recover();
            };
        }
      }
    }
  }
}

#endif
```

Program 5.7: Monitor class declaration

The OmniVision class processes the image data of the agent and provides usable data and information regarding the surrounding environment of the agent (see program 5.8 for the class declaration).

```
#ifndef PG_TRANSFERLAYER_ROBOT_OMNIVISIONOMNIVISION_H
#define PG_TRANSFERLAYER_ROBOT_OMNIVISIONOMNIVISION_H

#include "visiontunnel.h"
#include "geometry.h"

namespace PG{
  namespace SHIAI{
    namespace TransferLayer
    {
      namespace Robot
      {
        namespace OmniVision
        {
          class ClassObjectPolarCoordinate
          {
            public:
              double  Distance;
              double  Angle;
```

```
    bool     Visibility;
    void operator = (PG::SHIAI::GateLayer::VisionTunnel::
            StructVisionObject obj)
    {
      Distance    = obj.Distance;
      Angle       = CorrectSignedAngle(obj.Angle);
      Visibility  = true;
    }
};
struct StructVisionWorld
{
  ClassObjectPolarCoordinate Ball;
  ClassObjectPolarCoordinate TmPlayer[5][2];
  ClassObjectPolarCoordinate OpPlayer[5][2];
  ClassObjectPolarCoordinate TmGoalRight;
  ClassObjectPolarCoordinate TmGoalLeft;
  ClassObjectPolarCoordinate OpGoalRight;
  ClassObjectPolarCoordinate OpGoalLeft;
  ClassObjectPolarCoordinate TmOut;
  ClassObjectPolarCoordinate OpOut;
  ClassObjectPolarCoordinate LeftOut;
  ClassObjectPolarCoordinate RightOut;
  ClassObjectPolarCoordinate MidLine;
  ClassObjectPolarCoordinate TmRightCorner;
  ClassObjectPolarCoordinate TmLeftCorner;
  ClassObjectPolarCoordinate OpRightCorner;
  ClassObjectPolarCoordinate OpLeftCorner;
  ClassObjectPolarCoordinate TmLeftFlagPost;
  ClassObjectPolarCoordinate TmCenterFlagPost;
  ClassObjectPolarCoordinate TmRightFlagPost;
  ClassObjectPolarCoordinate Center;
  ClassObjectPolarCoordinate OpLeftFlagPost;
  ClassObjectPolarCoordinate OpCenterFlagPost;
  ClassObjectPolarCoordinate OpRightFlagPost;
  ClassObjectPolarCoordinate CoachChair;
  ClassObjectPolarCoordinate Obstacle[10][2];
  ClassObjectPolarCoordinate ReturnMarker;
  PG::SHIAI::GateLayer::VisionTunnel::StructHalfTimeColors  HalfTimeColor;
};
class ClassOmniVision
{
  public:
    ClassOmniVision();
    ~ClassOmniVision();
    bool Init();
    StructVisionWorld* Get();
    int LaunchVisionProcess(char* filename);
    bool KillVisionProcess();
    bool SetPallette(char* pallete, int pallettesize);
    bool SetHalfTimeColors(PG::SHIAI::GateLayer::VisionTunnel::
            GoalColors::Type goalcolor,PG::SHIAI::GateLayer::
```

```
                                VisionTunnel::TeamColors::Type teamcolor);
            private:
              PG::SHIAI::GateLayer::VisionTunnel::ClassVisionTunnel VisionTunnel;
              bool   ShowDebugInfo;
              int    MaxVisonWorldObjects;
            public:
              StructVisionWorld VisionWorld[1];
          };
        }
      }
    }
  }
}

#endif
```

Program 5.8: OmniVision class declaration

The OmniWheel class analyzes the movement decision vector and translates it into usable parameters to be sent to the motors to perform the move action. The reason this class is called OmniWheel is due to the fact that robot is equipped with three Swedish Omni Wheels that provide the robot with smooth omni-directional motion (see program 5.9 for the class declaration).

```
#ifndef PG_TRANSFERLAYER_OMNIWHEELOMNIWHEEL_H
#define PG_TRANSFERLAYER_OMNIWHEELOMNIWHEEL_H

#include "OmniWheelDefs.hpp"

namespace PG{
  namespace SHIAI{
    namespace TransferLayer{
      namespace Robot{
        namespace OmniWheel
        {
          class ClassOmniWheel
          {
          public:
            ClassOmniWheel(bool Simulate/*Configuration*/);
            ~ClassOmniWheel();
            bool SimpleMove(double velocity, double angle, double phi=0,
                      int ExecutionTime=-1) ;
            bool FastDribbleMove(double velocity, double angle,  double phi=0,
                        int ExecutionTime=-1) ;
            bool SingleFaceDribbleMove(double velocity, double angle,
                          RobotRegion::Type Region, double phi=0,
                          int ExecutionTime=-1) ;
            bool SimpleRotation(double r, double velocity, double phi,
                        int ExecutionTime=-1) ;
            bool RotateOnPoint(int vel, double currentradius, double desiredradius,
                      double currentangle, double desiredangle,
```

```
                double vtrain, double motorrefreshtime);
            bool Init();
            bool EnableMotors();
            bool Stop();
            bool DisableMotors();
            bool Face2Angle(double angle, double motorrefreshtime);
            bool RotateOnBall(double speed, RobotRegion::Type region,
                RotateDirection::Type dir);
            private:
              double SpeedUp(double vel, double MaxVelinit, double MaxVelFinal);
          private:
            ClassRobotSpeed DesiredSpeed;
            ClassRobotSpeed LastSpeed;
            ClassMotorSpeed MotorsSpeeds;
            ClassMotorSpeed OldWheelSpeeds;
            double WheelRadius;
            double sindelta;
            double cosdelta;
            double FrontWheelsDistace2Center; //meter
            double BackWheelDistace2Center;   //meter
            long   MaxWheelVelocity;          //cm/s
            double MaxDribblingMoveAngle;     //rad
            double MaxRotationalVelocity;     //rad/s
            double MaxRobotVelocity;          //cm/s
            double MaxPhi4DribbleOnMaxVel;    //rad/s
            double MaxPhi4DribbleOnMinVel;    //rad/s
            double MaxSimpleDribblingAngle;   //rad
            double AngleSmoothingCoefficent; //N.D.
            double RotateOnBallRadius;        //meter
            long   MaxWheelAcceleration;
            long   MaxWheelDeceleration;
            bool   ShowDebugInfo;
            bool   AutoChooseRealizableRobotVector;
            bool   SimulateRunning;
            public:
              ClassMotorControl Motor;
              void Rotate();
            private:
              ClassMotorSpeed ComputeRealizableSpeed( ClassRobotSpeed newspeed,
                              int executiontime,RobotRegion::Type region);
              ClassMotorSpeed ComputeMotorSpeeds(ClassRobotSpeed robotvec);
              ClassMotorSpeed RobotSpeedInverseKinematic(ClassRobotSpeed robotSpeed);
              ClassRobotSpeed RobotSpeedDirectKinematic(ClassMotorSpeed robotSpeed);
              ClassMotorSpeed CorrectMotorsSpeeds2Region(ClassMotorSpeed motor,
                                RobotRegion::Type region) ;
          };
        }
      }
    }
  }
}
```

```
#endif
```

Program 5.9: OmniWheel class declaration

The WorldMaker namespace contains one class which is responsible to process the data provided by the OmniVision such that the World Model Layer can use them to create the agent's world. This class allocates appropriate information for each defined object of the world and also collects data for the unknown objects tagging them as *unknown* (see program 5.10 for the class declaration).

```
#ifndef PG_TRANSFERLAYER_WORLDMAKERWORLDMAKER_H
#define PG_TRANSFERLAYER_WORLDMAKERWORLDMAKER_H

#include "omnivision.h"
#include "ball.h"
#include "goal.h"
#include "line.h"
#include "marker.h"
#include "obstacle.h"
#include "player.h"
#include "timer.h"
#include "IntranetWorld.h"
#include "visiontunnel.h"

namespace PG{
  namespace SHIAI{
    namespace TransferLayer{
      namespace WorldMaker
      {
        namespace BallOwner
        {
          enum BallOwners
          {
            Me,
            Teammate,
            Opponent,
            Invisible,
            Free,
            Undefined
          };
          typedef BallOwners Type;
        }
        namespace HalfTime
        {
          enum HalfTimes
          {
            First,
            Second,
            OverFirst,
            OverSecond,
```

```
    Penalty
  };
  typedef HalfTimes Type;
}
namespace KickerStatus
{
  enum KickerStatusType
  {
    Ready,
    Pending
  };
  typedef KickerStatusType Type;
}
struct StructPlayStatus
{
  PG::SHIAI::GateLayer::VisionTunnel::StructHalfTimeColors HalfTimeColor;
};
struct StructRobotStatus
{
  KickerStatus::Type KickerStatus;
};
class ClassWorldMaker
  {
  public:
    ClassWorldMaker();
    ~ClassWorldMaker();
    void Update(PG::SHIAI::TransferLayer::Robot::OmniVision::
          StructVisionWorld* visionworld, PG::Fundamentals::
          Timer::ClassTimer UpdateTime);
    void SetAllInvisible();
    PG::SHIAI::TransferLayer::Intranet::IntranetWorld::
      StructIntranetWorld GetIntranetWorld();
    PG::SHIAI::WorldObjects::ClassPlayer  TmPlayer[5];
    PG::SHIAI::WorldObjects::ClassPlayer  OpPlayer[5];
    PG::SHIAI::WorldObjects::ClassGoal    TmGoal;
    PG::SHIAI::WorldObjects::ClassGoal    OpGoal;
    PG::SHIAI::WorldObjects::ClassLine    TmOut;
    PG::SHIAI::WorldObjects::ClassLine    OpOut;
    PG::SHIAI::WorldObjects::ClassLine    LeftOut;
    PG::SHIAI::WorldObjects::ClassLine    RightOut;
    PG::SHIAI::WorldObjects::ClassLine    MidLine;
    PG::SHIAI::WorldObjects::ClassMarker  TmRightCorner;
    PG::SHIAI::WorldObjects::ClassMarker  TmLeftCorner;
    PG::SHIAI::WorldObjects::ClassMarker  OpRightCorner;
    PG::SHIAI::WorldObjects::ClassMarker  OpLeftCorner;
    PG::SHIAI::WorldObjects::ClassMarker  TmLeftFlagPost;
    PG::SHIAI::WorldObjects::ClassMarker  TmCenterFlagPost;
    PG::SHIAI::WorldObjects::ClassMarker  TmRightFlagPost;
    PG::SHIAI::WorldObjects::ClassMarker  Center;
    PG::SHIAI::WorldObjects::ClassMarker  OpLeftFlagPost;
    PG::SHIAI::WorldObjects::ClassMarker  OpCenterFlagPost;
```

```
            PG::SHIAI::WorldObjects::ClassMarker   OpRightFlagPost;
            PG::SHIAI::WorldObjects::ClassObstacle Obstacle[10];
            PG::SHIAI::WorldObjects::ClassMarker   CoachChair;
            PG::SHIAI::WorldObjects::ClassMarker   VirtualPoint;
            PG::SHIAI::WorldObjects::ClassBall     Ball;
            BallOwner::Type                         BallOwner;
            StructPlayStatus                       PlayStatus;
            StructRobotStatus                      RobotStatus;
          };
      }
    }
  }
}

#endif
```

Program 5.10: WorldMaker class declaration

5.3 The Decision Layer Implementation

The Decision Layer as the name implies is responsible to make decisions with respect to the goal of the system and status of the world. It has been discussed before that there are two layers of decision: High-Level Decisions and Low-Level Decision. In the soccer robots implementation the High-Level decision is consisted of three types of decisions: Strategic Decisions, Local Decisions, and Atomic Decisions. The Strategic Decisions are for instance attack, defend, etc. which are engaged for a particular period of the play for all robots and are usually triggered by the coach or, if exists, leader (captain) agent. Local Decisions are for instance GotoBall, LocalAttack, GotoGoal, etc. which are directly launched by the agent. The value for the electric charge of every object is allocated based on the Local Decisions. The Atomic Decisions are for instance Move, MoveNKick, Stop, MoveNHandleOn, etc. these decisions are considered to be the final output decision vector which are sent to the Transfer Layer. The Atomic Decision also determines the execution type of $\vec{F_T}$ (see equation 2.3. These decisions are made sequentially in a defined order. The sequence is based on the defined Decision Tree begins by making the Strategic Decision and following this decision a Local Decision will be made and finally the Atomic Decision will be made. For example, if the Strategic Decision is attack, and ball is free (it is in play and is not possessed by any player) the decision for players who see the ball would be as follows: Their first Local Decision is GotoBall, and for executing it the Atomic Decision would be MoveNHandleOn (move and turn on ball handling devices) will be launched and the path to reach the ball will be determined by the resultant vector of the electric fields vectors.

This layer also analyzes the predicted state of the world provided by the Predict Layer in order to make an optimum decision.

The Decision Layer namespace comprise two namespaces: HighLevelDecision and LowLevelDecision.

The HighLevelDecision contains two classes: LEFG and EFGeneratingHLD. The EFGeneratingHLD class makes High-Level decisions in terms of strategy (see program 5.11 for the class declaration).

```
#ifndef PG_SHIAI_DECISIONLAYER_HIGHLEVELEFGENERATINGHLD_H
#define PG_SHIAI_DECISIONLAYER_HIGHLEVELEFGENERATINGHLD_H

#include "efgeneratinghldDefs.hpp"

namespace PG{
  namespace SHIAI{
    namespace DecisionLayer{
      namespace HighLevelDecision{
        namespace Base{
          class ClassEFGeneratingHLD
            {
              public:
                ClassEFGeneratingHLD();
                ~ClassEFGeneratingHLD();
                MoveSkill::Type  GetMoveSkill();
                KickSkill::Type  GetKickSkill();

              public:
                StructHLD    HLD[1];
            };
        }
      }
    }
  }
}

#endif
```

Program 5.11: EFGeneratingHLD class declaration

The LEFG class is responsible to make High-Level decisions based on the strategy that is determined by the EFGenerateHLD (see program 5.12 for the class declaration).

```
#ifndef PG_SHIAI_DECISIONLAYER_HIGHLEVELDECISION_SELECTIVEBASEDLEFG_H
#define PG_SHIAI_DECISIONLAYER_HIGHLEVELDECISION_SELECTIVEBASEDLEFG_H

#include "worldmaker.h"
#include "efgeneratinghld.h"
```

```
using namespace PG::SHIAI::DecisionLayer::HighLevelDecision::Base;

namespace PG
  {
  namespace SHIAI
    {
    namespace DecisionLayer
      {
      namespace HighLevelDecision
        {
        namespace SelectiveBased
          {
          class ClassLEFG
            {
            public:
              ClassLEFG();
              ~ClassLEFG();
              void Generate(PG::SHIAI::TransferLayer::WorldMaker::ClassWorldMaker* world
                      Strategy::Type strategy);
            };
        }
      }
    }
  }
}

#endif
```

Program 5.12: LEFG class declaration

The LowLevelDecision contains one class that is responsible to make the Low-Level decisions and generate all the possible final decision vectors. It is also responsible to select the appropriate number of final vectors that have been generated (see program 5.13 for the class declaration).

```
#ifndef PG_DECISIONLAYERDECISION_H
#define PG_DECISIONLAYERDECISION_H

//#define XMLBased

#include "vector.h"
#include "worldmaker.h"

#ifdef XMLBased
    #include "xmlbasedefghld.h"
    using namespace PG::SHIAI::DecisionLayer::HighLevelDecision::XMLBased;
#else
    #include "selectivebasedefghld.h"
    #include "lefg.h"
    using namespace PG::SHIAI::DecisionLayer::HighLevelDecision::SelectiveBased;
#endif
```

```
using namespace PG::SHIAI::TransferLayer::WorldMaker;

namespace PG{
  namespace SHIAI{
    namespace DecisionLayer{
      namespace LowLevelDecision
      {
        class ClassDecision
          {
          public:
            ClassDecision(StructMemory *memory);
            ~ClassDecision();
            bool Make(ClassWorldMaker* World);
            double Speed(double maxspeed);
            PG::Fundamentals::Vector::ClassVector ComputeKickVector(
                    ClassWorldMaker* world, KickSkill::Type);
            PG::Fundamentals::Vector::ClassVector ComputeMoveVector(
                    ClassWorldMaker* world);
            PG::Fundamentals::Vector::ClassVector ComputeBHSVector(
                    ClassWorldMaker* world, KickSkill::Type);
            RobotRegion::Type GetBallOwnerRegion();
              MoveSkill::Type                          MoveSkill;
              PG::Fundamentals::Vector::ClassVector    MoveVector;
              PG::Fundamentals::Vector::ClassVector    KickVector;
              PG::Fundamentals::Vector::ClassVector    BHSVector;
            private:
#ifdef      XMLBased
              ClassXMLBasedEFGHLD                      EFGHLD;
#else
              ClassSelectiveBasedEFGHLD                SelectiveHLD;
              ClassLEFG                                LEFG;
#endif
            StructHLD                                  *HLD;
            StructMemory                               *Memory;
            double                                     MaxDecisionVector;
          };
      }
    }
  }
}

#endif
```

Program 5.13: Low-Level Decision class declaration

5.4 The World Model Layer Implementation

The World Model Layer has two major tasks: one is to create the world according to the method called as Agent Relative Polar Localization, and the second task is to generate the corresponding electric field vector for every object. This layer is defined as a namespace and to execute its allocated tasks have one class called ARPL.

The ARPL class retrieves all the world data and based on the available information calculates the angle and fuzzy distance of every object in the world relative to the agent. It also determines whether the important objects of the world, (i.e ball) are visible at the current processing cycle or not. This is because in the proposed soccer scenario the position of objects like ball is important as directly affects the decision making process of the agent. Therefore, if there is no information available for that object at a processing cycle, the ARPL informs the World Model Layer must inform the Decision Layer that this object is currently not visible. However, under such circumstances this class can provide approximate information regarding the position of the invisible object according to the predicted position of this object calculated in the Predict Layer (see program 5.14 for the class declaration).

```
#ifndef PG_SHIAI_ARPLARPL_H
#define PG_SHIAI_ARPLARPL_H

#define NON_VISIBLE      9999

namespace PG{
  namespace SHIAI{
    namespace ARPL{
      namespace FuzzyDistance
      {
        enum FuzzyDistances
        {
          Close,
          TooNear,
          SoNear,
          Near,
          Middle,
          SoFar,
          Far,
          TooFar,
          Infinite,
        };
        typedef FuzzyDistances Type;
      }
      class ClassARPL
        {
        public:
```

```
            ClassARPL();

            ~ClassARPL();
            void   operator += (ClassARPL arpl);
            void   operator =  (ClassARPL arpl);
            bool   operator <  (FuzzyDistance::Type dis);
            bool   operator <= (FuzzyDistance::Type dis);
            bool   operator >  (FuzzyDistance::Type dis);
            bool   operator >= (FuzzyDistance::Type dis);
            bool   operator == (FuzzyDistance::Type dis);
            bool   operator != (FuzzyDistance::Type dis);
            FuzzyDistance::Type C2Fuzzy(double distance);
            FuzzyDistance::Type C2Fuzzy();
            double C2Real(FuzzyDistance::Type distance);
    bool isVisible();
    double GetAngle();
    double GetDistance();
    double SetDistance(double dis);
    double SetAngle(double ang);
    void SetVisibility(bool status);
        private:
          double  Distance;
          double  Angle;
        };
    }
  }
}

#endif
```

Program 5.14: ARPL class declaration

5.5 The Predict Layer Implementation

The Predict Layer that is responsible to make predict the world state at a desired time. This layer is defined as a namespace and has one class to perform the prediction action (see program 5.15 for the class declaration).

```
#ifndef PG_SHIAI_TRANSFERLAYER_ROBOT_MONITORMONITOR_H
#define PG_SHIAI_TRANSFERLAYER_ROBOT_MONITORMONITOR_H

using namespace PG::SHIAI::TransferLayer::WorldMaker;

namespace PG
  {
  namespace SHIAI
    {
    namespace PredictLayer
      {
```

```
    class ClassPredict
    {
    public:
      ClassPredict();
      ~ClassPredict();
    Private:
      int Steps;
      int State;
      void Predict(int n, ClassWorldMaker* world);
    };
   }
  }
}

#endif
```

Program 5.15: Predict class declaration

5.6 The Implementation of the Object-Oriented Decision Tree

One of the most complex sections of the proposed system implementation for Soccer Robots is the implementation of the agent's main decision tree. In the very first version of the developed system there were a limited number of decisions for the agent, and in fact agent had only five Local Decisions and three Atomic Decisions which were eventually combined. Therefore, the coding implementation using *switch-case statements* seemed to be reasonable and efficient. However, further developments of the system led to the increase of the number of decision areas to six Strategic Decisions, twelve Local Decisions and five Atomic Decisions. Thereby, a rather huge condition area was formed. Anyhow, the new corresponding decision tree was also coded using *switch-case statements*. Further developments and complexity of the code urged to find a solution to simplify generating, correcting, tracing, and manipulating the decision tree. Mainly because the number of decision areas cannot be restricted to a specific numerical limit. The decision tree must be implemented such that it would be flexible and easy to add new decision areas even during agent's runtime. That is, the manipulation of the decision tree becomes of greater importance when allowing the agent or an outside source, such as coach, to modify the decision tree.

XML* was one of the best solutions to reach the above mentioned objectives. XML is very similar to the everyday language conversation where tree structures can be well implemented in Object-Oriented environments. Unlimited area of conditions has been made by generating an appropriate DOM†whilst only hav-

ing a limited code volume of XML. This method also provides the hereunder privileges:

- XML code is more human language friendly compared to programming languages such as C/C++
- XML code has a script nature structure. That is, any modification and correction does not require recompilation. This feature allows modifications to be made even during the system runtime.
- Independent programs or cross platform programs may use the same reference XML code. This becomes valuably important when it comes to robot learning.
- Due to the nature of Object-Oriented structure, the decision tree would be, if required, shown graphically for simpler manipulation.
- Implementation of learning algorithms for tracing or correcting the tree over a XML script is simpler and there is no need for having them within the agent's main program source code.

The agent traverses the decision tree at every decision making cycle. The traverse begins at the start point and goes through the conditions and eventually reaches a leaf where a decision is made.

*eXtensible Mark-up Language
†Data Object Model

Chapter 6

Conclusion

The main objectives of this thesis have been to:

1. Design a well-structured architecture for the system software
2. Design, develop, and implement a reliable and optimum algorithm for the intelligent agent to perform within a multi-agent system environment

A layered architecture was designed which was inspired by the OSI model. Thereby, each layer is independent and the connection between layers is through specific protocols.

This architecture significantly reduces the system upgrade costs because due to the independent nature of the layers, in case of an upgrade only the layers involved with the upgrade will need to be modified while the rest of the layers remain unchanged. This feature of the system software architecture well showed its benefits when during the implementation phase of the thesis the motor drivers were changed to completely new ones. The system was adapted and tested with the new motor drivers in only about two hours, while the same operation took about eight days for the other robots with the old system. The proposed architecture has made the overall performance of the system very reliable and robust. This is due to the fact that the layered architecture and the abstraction of each layer allows the Exception Handler to better recover from arising errors and malfunctions within the system performance. For example, if one of the system layers experience exceptions and errors it would not cause the whole system to crash, but only defected layer undergo the recovery process. This particular case have happened a couple of times for the Gate Layer as this layer was directly in contact with the physical world. The Gate Layer was crashed due to connection loss with the camera but the rest of the system remained alive while the Gate Layer recovered from the crash by restarting the camera interface.

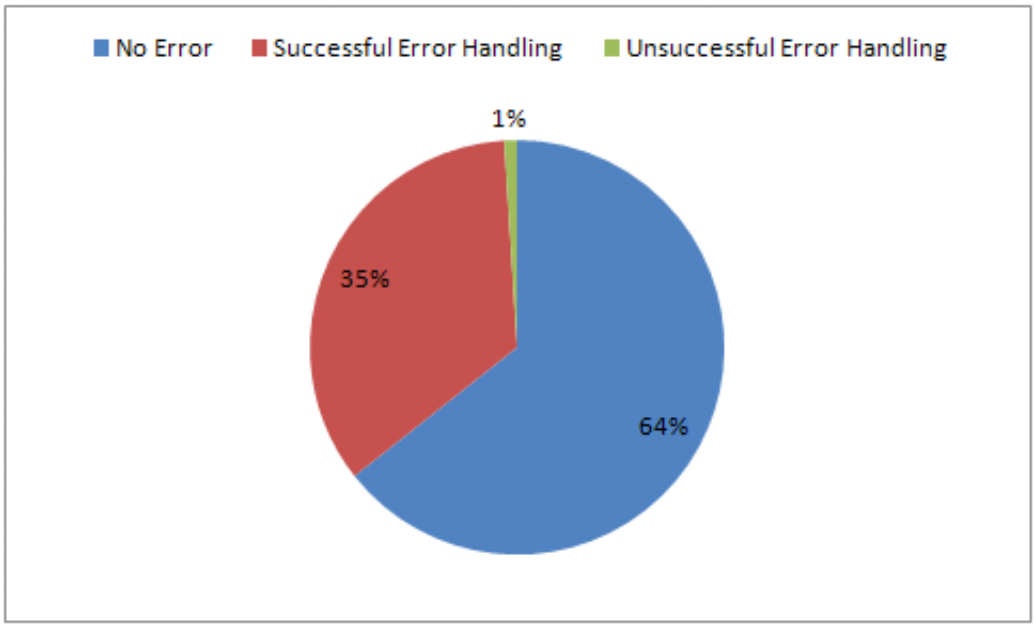

Figure 6.1: The performance of the system during totally 330 minutes runtime

The proposed intelligent algorithm in this thesis has three main sections: The electric fields vectors, the world model, the decision tree, and the learning part. The results of these sections have been discussed but it must also be mentioned that the combination of these methods have significantly reduced the amount of programming code volume and have also reduced the execution time making the system performance faster. For instance, in this thesis proposed method there is no need for additional programming code for dribbling, defending, and/or other similar actions. However, the resultant vector of the electric fields will lead the agent to perform such tasks in different situations.

The system was tested during the RoboCup 2010 MiddleSize League competitions. The testing was under tensed competitions circumstances. The robots that were run by this system were very reliable and robust throughout the competitions. The system never crashed except for one time which was caused by the power source failure, where the whole system was shutdown due to battery damage. There were, of course, errors and system exceptions during the runtime but all of them were successfully handled except for the one mentioned earlier. In overall, Figure 6.1 illustrates the system performance in terms of errors and exceptions throughout totally 330 minutes runtime.

Furthermore, it cannot be claimed the intelligence algorithms were 100% successful because there were some cases where better decisions could have been made. It is noteworthy that even though the intelligence performance of the sys-

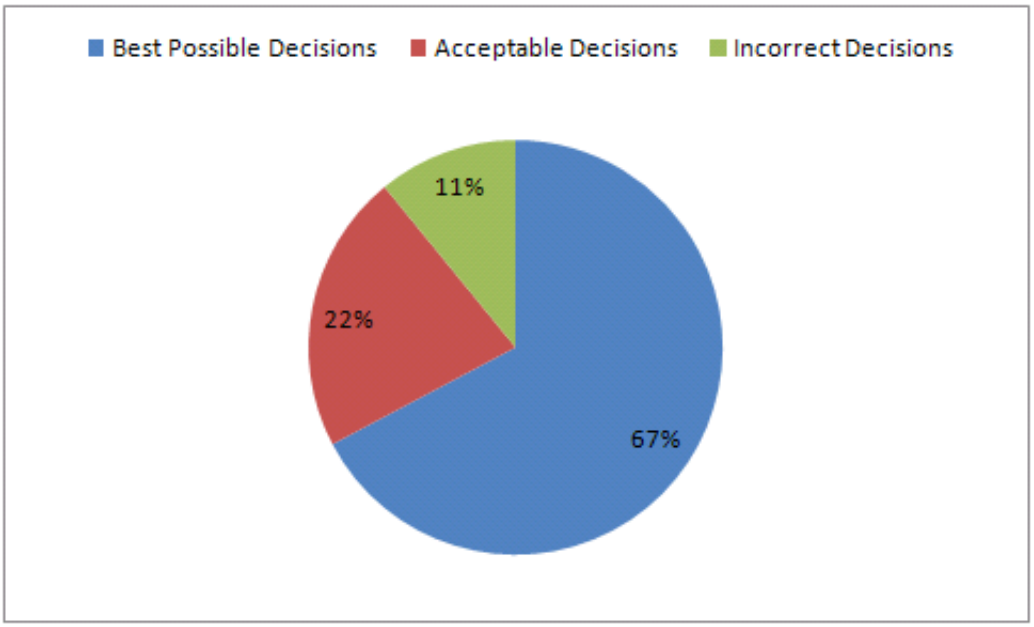

Figure 6.2: Evaluation of the decisions made by the system

tem was not of the most optimized one, but it has proven to be highly successful and reliable. Figure 6.2 shows the system performance in terms of decisions made in the 11 matches during the competitions. Totally 55 random decisions were selected for this evaluation.

It is strongly recommended for the future studies on this area to further develop the learning section of this system. Due to the complexity of learning and lack of time, it was not fully implemented during this thesis work but the studies discussed in this thesis would provide a good start point to further develop this section.

Bibliography

[1] J. Albus, "Outline for a theory of intelligence," *IEEE Trans. on Systems, Man, and Cybernetics*, vol. 21, no. 3, May/June 1991. [cited at p. 1]

[2] R. T. Laue, T., "A behavior architecture for autonomous mobile robots based on potential fields," *International RoboCup Symposium*, 2004. [cited at p. 3]

[3] H. Zimmermann, "Osi reference model-the iso model of architecture for open systems interconnection," *IEEE transactions on communications*, vol. com-28, no. 4, April 1980. [cited at p. 5]

[4] A. R. Meyer, J., "Decision-making and tactical behavior with potential fields," *Springer*, 2003. [cited at p. 9]

[5] B. M. Buck, Sebastian and T. Schmitt, "M-rose: A multi robot simulation environment for learning cooperative behavior," *Distributed Autonomous Robotic Systems*, vol. 4, 2002. [cited at p. 19]

[6] U. E. Asada, Minoru and K. Hosoda, "Cooperative behavior acquisition for mobile robots in dynamically changing real worlds via vision-based reinforcement learning and development," *Elsevier, Artificial Intelligence*, no. 110, pp. 275–292, 1999. [cited at p. 19, 23]

[7] N. Kubota and N. Aizawa, "Intelligent cooperative behavior control of multiple partner robots," *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2008. [cited at p. 19]

[8] H. M. Itsuki, Noda and H. Kazuo, "Learning cooperative behavior in multi-agent environment," *4th Pacific Rim International Conference on Artificial Intelligence*, 1996. [cited at p. 19, 23]

[9] S. S.-W. Lee, Dong-Wook and K.-B. Sim, "Online evolution for cooperative behavior in group robot systems," *International Journal of Control, Automation, and Systems*, vol. 6, pp. 282–287, 2008. [cited at p. 19]

[10] M. A. Dominey, Peter Ford and E. Yoshida, "Real-time cooperative behavior acquisition by a humanoid apprentice," *IEEE/RAS International Conference on Humanoid Robotics*, 2007. [cited at p. 19]

[11] A. B. R.S. Sotton, *Reinforcement Learning: An Introduction.* MIT Press, 1998. [cited at p. 19]

[12] P. S. Todd Hester, Michael Quinlan, "Generalized model learning for reinforcement learning on a humanoid robot," *IEEE International Conference on Robotics and Automation*, May 2010. [cited at p. 19]

[13] S. Haykin, *Neural Networks - A Comprehensive Foundation*, 2nd ed. Prentice Hall, 1999. [cited at p. 20]

[14] M. E. D'Angelo, A. and E. Pagello, "How a cooperative behavior can emerge from a robot team," *Springer*, 2004. [cited at p. 23]

Appendices

Appendix A

Appendix I - The ARICC

The Advanced Robotics and Intelligent Control Center (ARICC) aspire to be a center of excellence for R&D work in the areas of Advanced Robotics and Intelligent Control. ARICC aims to carry out research and development work in these areas through collaborative work with prestigious local and overseas universities and institutions by embarking on consultancy projects and initiating innovative projects in Singapore. The Center's aims include

- Undertaking the design, development, testing and commissioning of robots and intelligent systems for research, industry and education.
- Promoting R&D training of technologists in the areas of robotics and intelligent systems.

Technology areas/Industry of focus: The Center focuses on building up capabilities and expertise in areas such as humanoid robotics, service robots, educational robots, multi-agent collaboration, machine vision, real-time control, intelligent systems etc.

Appendix B

Appendix II - The RoboCup MiddleSize League Competitions

In the MiddleSize robot league, two teams of up to 5 robots play soccer on an 18x12 meter indoor field. Each robot is equipped with sensors and an on-board computer to analyze the current game situation and successfully play soccer.

Through wireless communication they can establish inter-team cooperation and receive all referee commands. But no external intervention by humans is allowed, except substitutions.

In recent years research made good progress. Just a year ago, the robots were only able to distinguish their own goal from the opponent goal by the goal color (the goals were colored yellow and blue respectively). At this year's tournament all teams were able to play with net goals only. The ball is the only item that is still color-marked.

Changes in the tournaments regulation like this will bring us eventually to the overall RoboCup goal:

By the year 2050, develop a team of fully autonomous humanoid robots that can win against the human world soccer champion team.

List of Symbols and Abbreviations

Abbreviation	Description	Definition
ARPL	Agent Relative Polar Localization	page 2
OSI	Open System Interconnection	page 5
GPS	Ground Positioning System	page 16
MDP	Markov Decision Process	page 19
XML	eXtensible Markup Language	page 49