A CONSTRAINT-BASED APPROACH TO REAL-TIME COOPERATIVE MULTIAGENT SYSTEMS: A SOCCER-PLAYING ROBOT TEAM

by

YU ZHANG

B.Eng. Zhejiang Institute of Technology, China, 1993

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF Master of Science

in

THE FACULTY OF GRADUATE STUDIES
(Department of Computer Science)

we accept this thesis as conforming to the required standard

The University of British Columbia
September 1998
© YU ZHANG, 1998
In presenting this thesis in partial fulfilment of the requirements for an advanced
degree at the University of British Columbia, I agree that the Library shall make it
freely available for reference and study. I further agree that permission for extensive
copying of this thesis for scholarly purposes may be granted by the head of my
department or by his or her representatives. It is understood that copying or
publication of this thesis for financial gain shall not be allowed without my written
permission.

Department of Computer Science
The University of British Columbia
Vancouver, Canada

Date 13 October, 1998
Abstract

It is not an easy task for a team of multiple fast-moving robots to cooperate with each other and try to beat another team in a dynamic, real-time environment. In order for a robot team to successfully play a soccer game, various technologies have to be incorporated including: robotic architectures, multi-agent collaboration and real-time reasoning.

A robot is an integrated system, with a controller embedded in its plant. A robotic system is the coupling of a robot to its environment. Robotic systems are, in general, hybrid dynamic systems, consisting of continuous, discrete and event-driven components. Constraint Nets (CN) provide a semantic model for modeling hybrid dynamic systems. Controllers can be embedded constraint solvers that solve constraints in real-time.

The controller for the robot soccer team player, UBC Dynamo98, has been modeled in Constraint Nets and implemented in Java. The various multi-agent collaborations have been treated as a part of the set of constraints which the controller solves in real-time.

A coach program using an evolutionary algorithm has also been designed and implemented to adjust the weights of the constraints and other parameters in the controller.
The results of the soccer tournament suggest that the formal Constraint Net approach is a practical tool for designing and implementing controllers for robots in multi-agent real-time environments. They also demonstrate the effectiveness of applying the evolutionary algorithm to the CN modeled controllers.
# Contents

Abstract ii  
Contents iv  
List of Tables viii  
List of Figures ix  
Acknowledgements x  

1 Introduction 1  
1.1 Motivation ............................................. 1  
1.2 Related Background and History .......................... 3  
1.3 The Problems and the Proposed Solutions ................. 4  
1.4 Outline of This Thesis .................................. 6  

2 A Constraint-based Approach to Real-time Cooperative Multi-agent Systems 7  
2.1 Constraint Nets model (CN) and Constraint-based Control .... 7  
2.1.1 Constraint Nets model (CN) .............................. 8  
2.1.2 Constraint-based Control ............................... 11
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.2</td>
<td>Constraint Nets (CN) Based Robot Architecture</td>
<td>12</td>
</tr>
<tr>
<td>2.3</td>
<td>Constraint-Based Control for Multi-agent Systems</td>
<td>15</td>
</tr>
<tr>
<td>2.3.1</td>
<td>Layered Embedded Constraint Solvers</td>
<td>16</td>
</tr>
<tr>
<td>2.3.2</td>
<td>Arbitration Hierarchy</td>
<td>16</td>
</tr>
<tr>
<td>2.3.3</td>
<td>Combined Utility Constraint</td>
<td>17</td>
</tr>
<tr>
<td>2.3.4</td>
<td>Evolutionary Algorithm</td>
<td>18</td>
</tr>
<tr>
<td>2.3.5</td>
<td>Special Constraints in a Multi-agent Environment</td>
<td>21</td>
</tr>
<tr>
<td>2.4</td>
<td>Discussion</td>
<td>23</td>
</tr>
<tr>
<td>3</td>
<td>A Multi-agent Testbed: the Soccer Server</td>
<td>25</td>
</tr>
<tr>
<td>3.1</td>
<td>System Overview</td>
<td>25</td>
</tr>
<tr>
<td>3.2</td>
<td>Field Simulator</td>
<td>28</td>
</tr>
<tr>
<td>3.3</td>
<td>Control Commands</td>
<td>28</td>
</tr>
<tr>
<td>3.4</td>
<td>Sensor Information</td>
<td>30</td>
</tr>
<tr>
<td>3.5</td>
<td>Coach Mode</td>
<td>31</td>
</tr>
<tr>
<td>3.6</td>
<td>Discussion</td>
<td>32</td>
</tr>
<tr>
<td>4</td>
<td>Designing and Programming the Controller for Soccer-playing Robots</td>
<td>35</td>
</tr>
<tr>
<td>4.1</td>
<td>The CN Architecture of the Controller for a Soccer-playing Robot</td>
<td>35</td>
</tr>
<tr>
<td>4.2</td>
<td>Constraint-Based Control for Soccer-playing Robot</td>
<td>40</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Constraint Methods in the Executor</td>
<td>40</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Constraint Methods in Planner</td>
<td>44</td>
</tr>
<tr>
<td>4.3</td>
<td>Java Implementation of the Constraint Based Controller</td>
<td>48</td>
</tr>
<tr>
<td>4.3.1</td>
<td>CN modules on Java Threads</td>
<td>48</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Event and Event Listener Classes</td>
<td>49</td>
</tr>
</tbody>
</table>
5 Designing and Programming the Coach Using an Evolutionary Algorithm

5.1 The Evolutionary Algorithm for the Soccer Team .......................... 52
5.2 The Design of the Coach ....................................................... 55
5.3 Java Implementation of the Coach ........................................... 58
  5.3.1 CN modules on Java Threads ........................................... 58
  5.3.2 The Individual and the Population Class ............................ 58
  5.3.3 Coach Class .............................................................. 59
5.4 Results .............................................................................. 59
5.5 Discussion ......................................................................... 60

6 Related Work ......................................................................... 61

6.1 Layered Learning ................................................................. 61
  6.1.1 Learning a Low-Level Multi-agent Behavior ......................... 62
  6.1.2 Using Decision Tree Confidence Factors for a Higher-level Deci-
  sion ............................................................................. 63
  6.1.3 Team-Partitioned, Opaque-Transition Reinforcement Learn-
  ing for Team-level Strategies .............................................. 66

6.2 Co-Evolving Soccer-playing Robot with Genetic Programming .... 66
6.3 Explicit Teamwork Model ...................................................... 68
6.4 Discussion ........................................................................... 68
List of Tables

4.1 Constraint based team vs. normal team .......................... 43

5.1 Evolved team from the 50th generation vs. Hand-tuned team .... 59
5.2 Evolved team from the 300th generation vs. Hand-tuned team ... 59
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>A robotic system</td>
<td>10</td>
</tr>
<tr>
<td>2.2</td>
<td>Abstraction hierarchy</td>
<td>13</td>
</tr>
<tr>
<td>2.3</td>
<td>Arbitration hierarchy (CS’s and A’s denote constraint solvers and arbiters respectively)</td>
<td>16</td>
</tr>
<tr>
<td>2.4</td>
<td>A general evolutionary algorithm EA</td>
<td>19</td>
</tr>
<tr>
<td>3.1</td>
<td>X windows interface of Soccer Server</td>
<td>26</td>
</tr>
<tr>
<td>3.2</td>
<td>Soccer Server Overview</td>
<td>27</td>
</tr>
<tr>
<td>4.1</td>
<td>The soccer-playing softbot system</td>
<td>36</td>
</tr>
<tr>
<td>4.2</td>
<td>The soccer-playing controller hierarchy</td>
<td>38</td>
</tr>
<tr>
<td>4.3</td>
<td>The player intercepts the ball, by setting different directions, the player can intercept the ball at two different positions when its speed is constant</td>
<td>43</td>
</tr>
<tr>
<td>4.4</td>
<td>The class inheritance hierarchy of visual objects</td>
<td>50</td>
</tr>
<tr>
<td>5.1</td>
<td>The Architecture of the Coach</td>
<td>56</td>
</tr>
</tbody>
</table>
Acknowledgements

I'm lucky to have Professor Alan Mackworth as my supervisor. His perspective of science and extremely wide knowledge in AI inspired me to explore the world of the unknown. Alan’s generous and continuous financial support is also greatly appreciated. He also provided funding for me to attend international conferences and competitions so that I could exchange and discuss research ideas with researchers from all over the world.

I'd especially like to thank my sister, Ying Zhang, and brother-in-law, Run-ping Qi, for their suggestions and help in my studies here, and for reviewing my thesis.

I thank the Dynamo group for their valuable suggestions on my thesis project.

I thank the Laboratory for Computational Intelligence (LCI) for their support of the genetic engineering on my soccer teams, which used twenty-four processes running on five Solaris PCs day and night. The system manager, Luk Dierckx, helped me a lot during my genetic engineering experiment.

I thank Michael Sahota for introducing me to the Dynamite robot soccer players.

Thanks also goes to my parents for their constant support and encouragement.
The University of British Columbia

September 1998
Chapter 1

Introduction

1.1 Motivation

The Good Old Fashioned Artificial Intelligence and Robotics (GOFAIR) [Mac93] research paradigm has shaped the area of intelligent robotics since the time of the robot Shakey. Some of the typical fundamental assumptions made about the world were that there is only one agent, that the environment is static unless the agent changes it, that actions are discrete and are carried out sequentially and that the world the robot inhabits can be accurately and exhaustively modeled by the robot. These assumptions proved to be overly restrictive and ultimately sterile. In the usual dynamic of the scientific dialectic, a new movement has emerged as a synthesis of GOFAIR and "Nouvelle AI": the Situated Agent approach. A situated agent is a real physical system grounded and embedded in a real world, here and now, acting and reacting in real-time. Mackworth [Mac93] proposed that playing soccer be a paradigmatic task domain since it breaks with nearly all of the restrictive assumptions on which GOFAIR is based and meets the requirements of the Situated
Agent approach. Chess is a traditional AI domain which has all of the restrictive assumptions of GOFAIR except two agents compete with each other. The soccer domain has the following characteristics compared to chess:

1. Multi-agent cooperation, communication and competition among friendly and hostile agents.

2. Real-time Dynamic environment. The environment changes while a robotic soccer player is deliberating.

3. Partial observable environment. A robotic soccer player’s sensors can not give it access to the complete state of the environment.

4. Nondeterministic environment. The next state of the environment can not be completely determined by the current state and the actions selected by the robotic soccer players.

5. Discrete/continuous hybrid environment. If there are a limited number of distinct, clearly defined perceptions and actions, we say that the environment is discrete. Sometimes a robotic soccer player can get distinct perceptions, like a referee’s commands, but also the speed and position of itself and other soccer players sweep through a range of continuous values.

With the recent achievement by the Deep Blue team, which beat Garry Kasparov, a human grand master, computer chess is close to its original goal.

Soccer as a test domain is sufficiently rich to support research integrating many branches of robotics and AI [SM94] and is suitable as a new challenge for the next generation AI.
1.2 Related Background and History

The Dynamite testbed has been developed in our laboratory for testing theories in the soccer domain using multiple mobile robots [BKM+93]. The testbed consists of a fleet of radio controlled vehicles that perceive the world through a shared overhead perceptual system. The Vision Engine produces the absolute position of all the objects on the soccer field. Each vehicle is controlled by a distributed user program running on two transputer nodes. The movement of all vehicles is controlled through radio transmitters attached to a single shared transputer node. A physics-based real-time graphics simulator for the Dynamite world is also available for testing and developing reasoning and control programs. Michael K. Sahota implemented a controller based on reactive deliberation to allow these robots to compete in one-on-one and two-on-two games of soccer. The robots can drive under accurate control at speeds up to 1 m/s, while simultaneously considering alternate actions [Sah94].

In June 1993, a group of researchers in Japan, including Minoru Asada, Yasuo Kuniyoshi, and Hiroaki Kitano, decided to launch a robotic competition, tentatively named the Robot J-League (J-League is the name of the newly established Japanese Professional soccer league). Within a month, they received overwhelming reactions from researchers outside of Japan, requesting that the initiative be extended as an international joint project. Accordingly, they renamed the project as the Robot World Cup Initiative, "RoboCup" for short. In September 1993, the first public announcement of the initiative was made, and specific regulations were drafted.

During the International Joint Conference on Artificial Intelligence (IJCAI-95) held at Montreal, Canada, August, 1995, the announcement was made to organize the First Robot World Cup Soccer Games and Conference in conjunction with IJCAI-97 Nagoya.
RoboCup consists of three competition tracks [KVM+97]:

1. **Real Robot League**: Using physical robots to play soccer games.

2. **Software Agent League**: Using software or synthetic agents to play soccer games on an official soccer server over the network.

3. **Expert Skill Competition**: Competition of robots which have special skills, but are not able to play a game.

To satisfy the need for a common environment, the Soccer Server was developed by Noda Itsuki [Itsb] to make it possible to compare various algorithms for multi-agent systems. Because the physical abilities of the players are all identical in the server, individual and team strategies are the focus of comparison. The Soccer Server is used by many researchers and has been chosen as the official simulator for the RoboCup Simulation League [Kit].

### 1.3 The Problems and the Proposed Solutions

Building control systems for autonomous robots working in a complex environment such as the soccer domain is an important challenge for research in AI.

A robot is an integrated system, with a controller embedded in its plant. A robot controller (or control system) is a subsystem of a robot, designed to regulate its behavior to meet certain requirements. A robotic system is the coupling of a robot to its environment. Robotic systems are, in general, hybrid dynamic systems, consisting of continuous, discrete and event-driven components. The dynamic relationship of a robot and its environment is called the behavior of the robotic system.
We have to face at least two questions involved in the design of robotic systems for soccer:

1. How to develop a robotic architecture in a multi-agent, real-time, dynamic, nondeterministic, inaccessible and discrete/continuous hybrid environment?

2. How to produce a controller so that the behavior of the robotic soccer system can meet certain requirements? As a robot soccer player in its soccer team, its requirement is to cooperate with its teammates to win the game.

The main goal of this thesis is to provide solutions for robot control in real-time multi-agent dynamic environments.

The solutions presented in this thesis are:

1. The Constraint Nets (CN) based robot architecture is developed for the robotic soccer system in a discrete/continuous hybrid environment. The Constraint Nets framework (CN) is a formal model for robotic systems and behaviours [ZM95a]. CN provides a theoretical foundation for systems design and analysis. The Constraint Nets framework captures the most general structure of dynamic systems so that systems with discrete and continuous time, discrete and continuous variables, and asynchronous as well as synchronous event structures can be modeled in a unitary framework [ZM]. CN provides aggregation operators to model a complex system hierarchically.

2. Constraint-based methods are used to produce a controller so that the behavior of the robotic soccer system can meet certain requirements. These requirements (including multi-agent cooperation requirements) impose global constraints over a system's state variables. We set weights for these constraints
and combine these constraints into a utility function, which assigns a single number to express the desirability of an action. The controller chooses the action with the highest utility, then maximizes the probabilities of satisfying all the constraints.

3. A coach program is designed and implemented to use an evolutionary algorithm to adjust these weights and other parameters in the controller.

1.4 Outline of This Thesis

This thesis is organized as follows: Chapter 2 discusses our approach to the problem — building the controller for soccer-playing robots via constraint-net-based robot architecture and constraint-based control. Chapter 3 describes the soccer server, a multi-agent testbed. Chapter 4 describes how to design and implement a controller for soccer-playing robots in Java. Chapter 5 describes how to design and implement a coach for soccer-playing robots using an evolutionary algorithm. Chapter 6 gives a survey of related work on multi-agent robot systems and soccer-playing robots. Chapter 7 summarizes our current work and points out the directions for further research.
Chapter 2

A Constraint-based Approach to Real-time Cooperative Multi-agent Systems

In this chapter, we present a constraint-based approach to real-time cooperative multi-agent systems. We start with the concepts of the constraint net model (CN) and constraint-based control. We then present a constraint-net-based robot architecture for real-time, dynamic and discrete/continuous hybrid environment. Then the constraint-based multi-agent control and its evolutionary algorithm are presented.

2.1 Constraint Nets model (CN) and Constraint-based Control

Ying Zhang and Alan Mackworth developed a semantic model for dynamic systems, Constraint Nets, in order to establish a foundation for modeling, specifying and
verifying discrete/continuous hybrid systems. **Constraint-based Control** was also developed by them as an integrated approach to the design and analysis of robotic systems and behaviors [Zha94].

### 2.1.1 Constraint Nets model (CN)

A robot is an integrated system, with a controller embedded in its plant. A robot controller (or control system) is a subsystem of a robot, designed to regulate its behavior to meet certain requirements. A robotic system is the coupling of a robot to its environment. Robotic systems are, in general, hybrid dynamic systems, consisting of continuous, discrete and event-driven components. The dynamic relationship of a robot and its environment is called the behavior of the robotic system.

Constraint Nets (CN), a semantic model for hybrid dynamic systems, can be used to develop a robotic system, analyze its behavior and understand its underlying physics. Using this model, we can characterize the components of a system and derive the behavior of the overall system. CN is an abstraction and generalization of data-flow networks. Any (causal) system with discrete/continuous time, discrete/continuous (state) variables, and asynchronous/synchronous event structures can be modeled. Furthermore, a system can be modeled hierarchically using aggregation operators; the dynamics of the environment as well as the dynamics of the plant and the controller can be modeled individually and then integrated [ZM].

A constraint net consists of a finite set of locations, a finite set of transductions and a finite set of connections. Formally, a **constraint net** is a triple $CN = \langle Lc, Td, Cn \rangle$, where $Lc$ is a finite set of locations, $Td$ is a finite set of labels of transductions, each with an output port and a set of input ports, $Cn$ is a set of connections between locations. A location can be regarded as a wire, a channel,
a variable, or a memory cell. Each transduction is a causal mapping from inputs to outputs over time, operating according to a certain reference time or activated by external events.

Semantically, a constraint net represents a set of equations, with locations as variables and transductions as functions. The *semantics* of the constraint net, with each location denoting a trace, is the least solution of the set of equations [Zha94, page 48]. Intuitively, a trace denotes changes of values over time. Formally, a mapping $v: T \rightarrow A$ from time $T$ to domain $A$ is called a *trace* [Zha94].

Given $CN$, a constraint net model of a dynamic system, the abstract behavior of the system is the semantics of $CN$, denoted $[CN]$, i.e., the set of input/output traces satisfying the model.

A complex system is generally composed of multiple components. A *module* is a constraint net with a set of locations as its interface. A constraint net can be composed hierarchically using modular and aggregation operators on modules. The semantics of a system can be obtained hierarchically from the semantics of its subsystems and their connections.

A constraint net is depicted by a bipartite graph where locations are depicted by circles, transductions by boxes and connections by arcs. A module is depicted by a box with rounded corners.

Furthermore, the environment of the robot can be modeled as a module as well. A robotic system can be modeled as an integration of a plant, a controller and an environment (Fig. 2.1). A plant is a set of entities which must be controlled to achieve certain requirements, for example, a car with throttle and steering. A controller is a set of sensors and actuators, which, together with software/hardware computational systems, (partially) senses the states of the plant ($X$) and the envi-
A robotic system computes the desired control inputs ($U$) to actuate the plant. An environment is a set of entities beyond the (direct) control of the controller, with which the plant may interact. For example, obstacles to be avoided and objects to be reached can be considered as the *environment* of a robotic system.

In most cases, desired goals, safety requirements and physical restrictions of a robotic system can be specified by a set of constraints on variables $U \cup X \cup Y$. The controller is then synthesized to regulate the system to satisfy the set of constraints. The semantics (or behavior) of the system is the solution of the following equations:

\[
X = PLANT(U, Y), \tag{2.1}
\]
\[
U = CONTROLLER(X, Y), \tag{2.2}
\]
\[
Y = ENVIRONMENT(X). \tag{2.3}
\]

Note that *PLANT*, *CONTROLLER* and *ENVIRONMENT* are transductions mapping input traces to output traces, and the solution gives $X$, $Y$ and $U$ as tuples of traces.
2.1.2 Constraint-based Control

Control systems are designed to meet certain requirements. A requirements specification $\mathcal{R}$ for a system $CN$ is a set of allowable input/output traces of the system: the behavior of $CN$ satisfies its requirements specification $\mathcal{R}$, written $[CN] \models \mathcal{R}$, iff $[CN] \subseteq \mathcal{R}$.

The problem of control synthesis can be formalized as follows: given a requirements specification $\mathcal{R}$, the model of the plant $PLANT$ and the model of the environment $ENVIRONMENT$, synthesize a model of the controller $CONTROLLER$, such that $[CN] \models \mathcal{R}$ where $CN$ is modeled by Eqs. (1), (2) and (3).

Ying Zhang and Alan Mackworth consider constraints as relations on a set of state variables; the solution set of the constraints consists of the state variable tuples that satisfy all the constraints. They call the behavior of a dynamic system constraint-based if the system is asymptotically stable at the solution set of the given constraints, i.e., whenever the system diverges because of some disturbance, it will eventually return to the set satisfying the constraints [ZM95b].

Most robotic systems are constraint-based, where the constraints may include physical limitations, environmental restrictions, and safety and goal requirements. Most learning and adaptive dynamic systems exhibit some forms of constraint-based behaviors as well.

A controller is an embedded constraint solver if the controller, together with the plant and the environment, satisfies the given constraint-based specification. Given a constraint-based requirements specification, the design of the controller becomes the synthesis of an embedded constraint solver which, together with the dynamics of the plant and the environment, solves the given constraints on-line.
2.2 Constraint Nets (CN) Based Robot Architecture

Ying Zhang and Alan Mackworth propose two kinds of hierarchy in a robot control system: one is the composition hierarchy, the other is the interaction hierarchy.

The Constraint Net model supports composition hierarchies with modules, which have a set of inputs and outputs and perform a transduction from input traces to output traces. The *composition hierarchy* characterizes the hierarchy of composing complex modules from simple ones. A complex module can be incrementally composed of simpler ones. A system can be tested and verified structurally.

The *interaction hierarchy* imposes the hierarchy of interaction or communication between modules. A control system is modeled as a module that may be further decomposed into a hierarchy of interactive modules (Fig. 2.2). The higher levels are typically composed of event-driven transductions and the lower levels are typically analog control components. The bottom level sends control signals to various effectors, and at the same time, senses the state of effectors. Control signals flow down and the sensing signals flow up. Sensing signals from the environment are distributed over levels. Each level is a black box that represents the causal relationship between the inputs and the outputs. The inputs consist of the control signals from the higher level, the sensing signals from the environment and the current states from the lower level. The outputs consist of the control signals to the lower level and the current states to the higher level. Usually, the bottom level is implemented by analog circuits that function with continuous dynamics and the higher levels are realized by distributed computing networks.

This CN architecture is object-oriented, parallel, and event-flow based. Events are used for communication among these CN modules. An event is an object which contains signals and (or) data. There are two types of CN modules, *event-driven* or
Figure 2.2: Abstraction hierarchy
fixed-sample-time-driven.

An event-driven CN module does nothing until it is woken by one or more events sent by other CN modules. Then it adjusts its states according to the signals in those event objects and processes the data in those event objects. After it finishes processing, it may produce its own events which contain the processed data, and send these events to wake other CN modules. Then the module goes to sleep again.

We define two kinds of input ports, alarm input ports and non-alarm input ports. Events that arrive on alarm input ports can wake up the CN module. Events that arrive on non-alarm input ports can not wake up the CN module, they are only treated as transports for data. An event-driven CN module can only be woken when events are present on all of the alarm input ports.

A fixed-sample-time-driven CN module works and rests according to a module independent fixed time schedule. It has only non-alarm input ports. When it is time to work and there are events waiting to be processed, the module gets the events and processes the data in them. When it is time to sleep, no event can wake it up.

Execution of a CN module causes the events on the input ports to be removed and new events to be produced on the output ports. When two or more events come to an input port, the old event might get kicked out of the port by the new event. This also depends on the type of input port. As for buffering, there are three types of input port:

1. Non-buffered input port. It can only support one event. The newly arrived event will kick out the old event.

2. Buffered input port. It supports an event queue. The newly arrived event waits behind the old event to be processed.
3. **Hybrid** input port. The event itself decides its own fate. If an event belongs to an event sequence, it will carry the information telling the input port that it is a member of an event sequence and the number of events that follows. The port will buffer it if it's in an event sequence so it can't be discarded by the new one. When it indicates that itself is an independent event, the new event still can kick it out of the port.

In a highly reactive real-time dynamic environment like the soccer domain, *non-buffered* or *hybrid* input ports are highly recommended. The agent has no time to waste on the out-of-date situations, it must react quickly to the newly arrived situations. The hybrid input port is more suitable for real world problems since sometimes events are related to each other and it is better to let the event itself decide its importance.

### 2.3 Constraint-Based Control for Multi-agent Systems

We first discuss the layered constraint-based control corresponding to the CN-based robot architecture. Then we describe the coordination among different constraint solvers at the same level of the interaction hierarchy. The concept of combined utility constraints and their utility functions is then discussed. Then, an evolutionary algorithm used for tuning the weights in the utility function and optimizing the performance of the whole controller is explained. Some special constraints existing in a multi-agent environment are described in the final subsection.
2.3.1 Layered Embedded Constraint Solvers

In the framework for control synthesis, constraints are specified at different levels on different domains, with the higher levels more abstract and the lower levels more plant-dependent [ZM].

A control system can also be synthesized as a hierarchy of interactive embedded constraint solvers. Each abstraction level solves constraints on its state space and produces the input to the lower level. Typically the higher levels are composed of digital/symbolic event-driven control derived from discrete constraint methods and the lower levels embody analog control based on continuous constraint methods.

2.3.2 Arbitration Hierarchy

In the framework of control synthesis, constraint solvers at the same level of the interaction hierarchy are coordinated via various arbitrations to compromise among different kinds of constraint, which form the arbitration hierarchy.

![Arbitration Hierarchy Diagram](image)

Figure 2.3: Arbitration hierarchy (CS’s and A’s denote constraint solvers and arbiters respectively)

One type of arbitration can be modeled by the subsumption architecture.
An output of a module in a higher layer (U) can be *subsumed* by an output of a module in a lower layer (L):

\[
f_s(L, U) = \begin{cases} 
  L & \text{if } L \neq 0 \\
  U & \text{otherwise.}
\end{cases} \tag{2.4}
\]

We can define some other arbitration functions [Zha94]:

- **Conditional pass:**
  \[
  f_c(C, I) = \begin{cases} 
    I & \text{if } C \neq 0 \\
    0 & \text{otherwise.}
  \end{cases} \tag{2.5}
  \]

- **Compromise:**
  \[
  f_w(I_1, I_2) = w_1 I_1 + w_2 I_2, \quad w_1, w_2 > 0, \quad w_1 + w_2 = 1. \tag{2.6}
  \]

In most cases, arbitration functions are nonlinear.

### 2.3.3 Combined Utility Constraint

Another way to deal with various constraints at the same level of the interaction hierarchy is to combine these constraints into one utility constraint. This combined utility constraint is to maximize the *utility function*:

\[
U(o) = \sum_i k_i * P_i(o) \tag{2.7}
\]

\(U(o)\) is the action \(o\)'s utility. \(P_i(o)\) is the probability of satisfying the constraint \(i\) when taking the action \(o\). \(k_i\) is the weight for the constraint \(i\).

The constraint solver for this combined utility constraint will output the action \(o\) with the highest utility. These weights can be set by hand. They can also
be tuned by a learning method, such as reinforcement learning. Also the utility function $U(a)$ need not be linear; it might be obtained by using neural network learning.

2.3.4 Evolutionary Algorithm

In this thesis, an evolutionary algorithm (EA) is used to adjust the weights in the utility function and other parameters in the controller.

Nature has a robust way of evolving successful organisms. A variety of evolutionary algorithms (EAs) have been proposed based on this successful natural process [HB98]. Generally speaking, EAs maintain a population of individuals that evolve according to the rules of selection and reproduction. There are several choices for what the individuals can be. They can be whole agent functions, or components of the whole agent functions. Each individual in the population receives a measure of its fitness in the environment. The highly fit individuals are selected and given rights to reproduce and the ill-suited individuals die off. Recombination and mutation are applied to those successful individuals to produce their offspring, thus providing general heuristics for exploring the available fitness information. Since the evolutionary process learns an agent function based on some special rewards like the rights to survive and to produce offspring, it can be seen as a form of reinforcement learning.

A general evolutionary algorithm EA [HB98] is shown in Fig. 2.4.

For applying EA to a problem, we have to answer the following three questions:

- How do individuals reproduce?
- How is an individual represented?
Algorithm EA

//start
\( t := 0; \)

//initialize a random population of individuals
\( P = \text{initpopulation}(t); \)

//evaluate fitness of individuals in population
\( \text{evaluate}(P); \)

//test for termination condition (time, fitness, etc.)
while not done do
    //increase the time counter
    \( t := t + 1; \)
    //select parents
    \( \text{PARENTS} := \text{select}(P, t); \)
    //reproduction
    \( \text{OFFSPRING} := \text{reproduce}(\text{PARENTS}); \)
    //evaluate the offspring’s fitness
    \( \text{evaluate}(\text{OFFSPRING}); \)
    //select the survivors to form the new generation
    \( P := \text{survive}(P, \text{OFFSPRING}); \)
end while
end EA

Figure 2.4: A general evolutionary algorithm EA
• How are the fit individuals selected?

There are two kinds of reproduction in nature, sexual or asexual. Asexual (cloning) reproduction typically results in offspring that are almost genetically identical to the parent. By saying "almost," it means there is a tiny chance of mutation. Large numbers of low-level organisms reproduce asexually; this includes most bacteria which some biologists hold to be the most successful species known. Evolutionary Programming (EP) is a kind of EA algorithm using asexual reproduction [HB98, Fog95].

Sexual reproduction allows some shuffling of chromosomes, producing offspring that contain a combination of information from each parent.

Why do all of the advanced organisms in nature reproduce sexually [Den95, Jon93]? There are at least two reasons. First, it allows reproduction based on individuals' fitness. To a certain degree, the next generation's quality is ensured. This is very important to advanced species which do not have the huge population that primitive species usually have.

Second, due to the low mutation rate in nature, if the advanced organisms reproduce asexually, there will not have much variety within the species because the population is not large. A species which lacks variety has a greater risk of being wiped out by the changing environment. The population of primitive species like bacteria or viruses is huge, so the mutation rate in nature is enough high to allow for variety in the population. That is why some viruses like the flu virus can still affect the human race from time to time no matter how effective the anti-virus medicine.

Based on these reasons, our EA uses sexual reproduction.

In our EA approach, an individual is represented as a pair of integer vectors. A vector is called a chromosome. An integer in the vector is called a gene which
partly decides one of the parameters in the controller. Each of the two chromosomes carries the same characteristic set of genes in a specific order. The sum of each pair of genes decides one parameter in the controller.

Two steps are needed to recombine the genes of the selected parents:

1. Each parent randomly chooses one gene from each pair of genes to form its sex chromosome.

2. Offspring is formed by receiving one sex chromosome from each of its parents and pairing them together.

Our EA uses tournaments to select fit individuals because of the competitive nature of the soccer domain. Each trial solution in the population faces competition against a preselected number of opponents and receives a "win" according to the rules of the game. Selection then eliminates those solutions with the least wins.

By this evolutionary algorithm, the constraint-based controller for the robot evolves to better satisfy the constraint requirements from the environment.

2.3.5 Special Constraints in a Multi-agent Environment

In the real world there are many tasks which can only be achieved successfully by a group of agents acting together. There is increasing interest in the issue of group co-operation amongst agents in order to more effectively achieve tasks. As robots become more adept at operating in the real world, the issues of collaborative and adversarial planning and learning are becoming more important. Here is an example in the robotic soccer domain to explain why we need collaborative systems. If there is no cooperation between robotic soccer players, teammates often attempt the same action and get in each others way. On the contrary, if we implement cooperation
between robotic soccer players, keeping alert and constantly studying the position of opponents and teammates will turn a wild aimless kicking game into a smooth, beautiful passing attack.

Based on the Constraint Nets (CN) model, we simply treat other agents as a part of the environment which is modeled as Eqs. (3). This multi-agent part of the environment also provides some constraints on the agent as the rest of the environment does.

These multi-agent constraints can be classified into three types:

1. **Role Constraints.** Because of limited abilities of a single agent, each agent plays a separate role within the team so the whole team can achieve the goal or achieve it more efficiently.

2. **Social Law Constraints.** A simple way to provide coordination among agents is social laws or conventions. These social laws are the rules that define common actions which any agent has to do under certain conditions. If these social laws or conventions are obeyed by all the agents, no conflicts can happen and the common goal might be achieved. Social law constraints are very useful when different cooperative agents conflict with each other while trying to perform their roles.

3. **Communication Constraints.** A team of agents can also communicate with each other to coordinate their actions. This is especially effective when the environment is partially observable by the agents. An agent may not have a reasonable action when it can only get partial information. Another agent situated somewhere else might have a better view of the whole situation and think its teammate has to do certain actions under that condition. So it
communicates with its teammate. The agent gets the information from its teammate, it can do its own actions or act as its teammate suggests. This decision depends on their ranks in the team and the confidence factor of the commands they communicate.

2.4 Discussion

CN based robot architecture is concurrent and object-oriented. Each module in the CN is an object. These modules only communicate with each other using events, which are also objects. This concurrent architecture reflects the distributed and concurrent nature of real-time dynamic complex systems (including non man-made systems, such as organisms). Because of this feature, it is natural for us to use object oriented methodology and programming language which is very useful for programming a large and complex system. It is also natural to run such systems on a concurrent and distributed computer system to improve real-time performance and efficiency.

CN based robot architecture also reflects the hierarchical nature of real-time dynamic complex systems. The universe itself is a perfect example of a highly hierarchical system. A complex module in CN can be incrementally composed of simpler ones. Any simple module can run on a CPU, so the improvement of real-time performance and efficiency is only limited by the computing resources.

The multi-agent constraints are mainly caused by the limited abilities of a single agent who faces a complex task. When other agents come to help, a set of rules and communication is needed to prevent conflicts and improve efficiency.

Our constraint-based approach strongly reflects the golden rule for dealing with the complex real world, that is, "Divide and Conquer". Our Evolutionary
algorithm which is borrowed from nature is also supposed to be an effective way to evolve a successful system for dealing with a complex real-time dynamic environment.
Chapter 3

A Multi-agent Testbed: the Soccer Server

Noda Itsuki developed Soccer Server, a simulator of soccer games, to provide a common test bed to evaluate various multi-agent systems. Using this soccer server, a team of players written in any kind of programming language with facilities of UDP/IP can play with another team written in a different language. A controller for our soccer players playing on the soccer server, Dynamo98, has been developed using the constraint based approach, and implemented in Java. Some important features of the soccer server are mentioned here. For more details about how soccer server works, please see Appendix A.

3.1 System Overview

Soccer Server provides a virtual soccer field in which players controlled by clients run and kick a ball (Fig. 3.1).

The soccer Server consists of 3 modules: a field simulator module, a referee
Figure 3.1: X windows interface of Soccer Server
module and a message-board module (Fig. 3.2). The field simulator module calculates movements of objects on the field and checks collisions among them. The referee module controls a game according to rules. The message-board module manages communication among clients.

A client connects with the server by a UDP socket. The server assigns a player to the client. Using the socket, the client sends commands to control its player and receives information from sensors of the player. Each client controls only one player. The soccer server connects with 22 clients at most, so each team can have 1 to 11 players. One of these players in each team can be a goalie. All communication between the server and each client is done by using ASCII strings. The protocol of the communication consists of control commands and sensor information.
3.2 Field Simulator

The soccer field and all objects on it are two-dimensional. The nominal size of the field is the same as the official size of a human soccer field. The length is 105 meters and the width is 68 meters. The width of the goals is doubled to 14.64 meters.

Players and the ball are treated as circles. Movements of these objects are simulated stepwise one by one for every 100 milliseconds (ms). The velocity of an object is added to its position, while the velocity decays at a known rate and increases according to its acceleration given by commands from clients. Noise is added to each movement according to the speed of the object. If an object overlaps another object, that is, collides with another object after its movement, the object is moved back until it does not overlap other objects. Then its velocity is multiplied by -0.1.

The simulator also provides various landmarks, which are flags, lines and goals. These landmarks are visible to the players and do not move. The simulator does not provide absolute positions of the players to the clients, so clients need to determine their players’ positions from the relative positions of these fixed landmarks.

3.3 Control Commands

Control commands are messages sent from a client to the server to control actions of the client’s player. There are seven control commands, turn, dash, kick and catch are used for controlling the player’s physical action during a match. say is used for communication among players. change_view is used for changing the mode of the visual sensor. move is used to move the player to a certain position before kick-off.
1. (move X Y). Move the player to the position (X,Y).

2. (turn Moment). Change the direction of the player according to Moment. Actual change in the direction is reduced when the player is moving quickly.

3. (dash Power). Increase the velocity of the player in the direction it’s facing by Power. Each player has a certain stamina that decreases when the player dashes, and recovers it with every step until it reaches the maximum. Longer dashes cause stamina to be regained slowly. In this model the long term effect of a dash reduces the player’s ability to recover.

4. (kick Power Direction). Kick the ball by Power in the Direction if the ball is near enough.

5. (catch Dir). This command is used only by the goalie to catch the ball which falls in the small catchable rectangular area in front of the goalie along Dir. When this command is executed, all opponents who are near the ball will be pushed 9 meters away from the ball. Only one goalie is available in a team.

6. (say Message). Broadcast the Message to all players. Message must be an ASCII string whose length is less than 512.

7. (change_view Angle_width Quality). Change angle of view cone and quality of visual information. Angle_width can be one of wide (180 degrees), normal (90 degrees) or narrow (45 degrees). Quality can be one of high or low. The server sends detailed information about positions of objects to the client. Only reduced information (directions) of objects are sent to the client in the case of low quality.
3.4 Sensor Information

The server sends two kinds of sensor information to the client. They are:

1. Visual Information, (see Time ObjInfo ObjInfo ...). *Time* indicates the current time. *ObjInfo* is information about a visible object, whose format is: *(ObjName Distance Direction DistChng DirChng)*. *Distance* and *Direction* are relative distance and direction of the object respectively, and *DistChng* and *DirChng* are respectively the radius and angle components of change of the relative position of the object. The visual objects are players with their team name and number, ball, flags, goals and lines. As the distance to a player becomes larger, more and more information about the player is lost. First players’ numbers can not be seen further than a certain distance, then players’ team names are lost when they are too far away. The quality of visual information is also changed by change_view command. The frequency of sending this information depends on the quality of information transmitted. When the view cone angle is normal and the quality is high, the information is sent every 150 ms. When the angle changes to wide the frequency is halved, and when the angle changes to narrow the frequency is doubled. When the quality is low, the frequency is also doubled. The player can ‘see’ objects in the view cone or within 3 meters. When the object is within 3 meters but out of the view cone, the player can only identify the type of object (ball, player, goal or flag), but not the exact name of the object. The value of distance to the object is quantized in a certain method so that the noise in the vision information increases when the object moves away from the player.

2. Auditory information. (hear Time Direction Message). All communication
among clients is done via the server. Direct communication is not permitted. 

_Direction_ is the direction of the sender. In the case that the sender is the client itself, _Direction_ is 'self'. _Time_ indicates the current time. This message is sent immediately when a client sends a (say _Message_) command. The server only accepts one message from each client for every simulation step. A player can hear only one message from each team during two simulation steps. The server chooses the message to be broadcast according to the order of arrival. And a message said by player-X is informed only to players within 50 meters from player-X. Because of these restrictions, the reliability of communication is not guaranteed. Judgments of the referee are also broadcast using this form. In this case, _Direction_ is 'referee'. Possible judgments are: _before_kick_off_, 

_kick_off_l_ (kick off by the left side), _kick_off_r_, _kick_in_l_, _kick_in_r_, _corner_kick_l_, _corner_kick_r_, _goal_kick_l_, _goal_kick_r_, _free_kick_l_, _free_kick_r_, _play_on_, _half_time_, _time_up_.

### 3.5 Coach Mode

In order to make it easy to develop client programs, the server provides 'coach mode'. If the soccer server is invoked with the '-coach' option, the server runs in coach mode. In coach mode, it prepares a UDP socket bound to the coach port (default 6001). A privileged client (called 'coach client') uses this coach port and takes the place of the referee module in the server. The coach client can control play mode, move objects (players and a ball), get information about positions of all objects on the field and broadcast a message as a judgment of the referee. The commands of coach are:
1. (move $OBJECT \ X \ Y [VDir]$). Move $OBJECT$ (players or a ball) to the position $(X,Y)$, and change the direction of $OBJECT$ to $VDir$.

2. (look). Get information about positions of all objects on the field. The server returns (look $TIME \ OBJ1 \ X1 \ Y1 \ OBJ2 \ X2 \ Y2$...).

3. (check_ball). Check the position of the ball. The server returns (ok check_ball $TIME \ BALL-POSITION$). $BALL-POSITION$ can be one of in_field, goal_l, goal_r, or out_of_field.

4. (ear $MODE$). Turn on or off sending auditory information. $MODE$ must be on or off. If on, the server sends all auditory information to the coach client in the same manner as to player clients. If off, the server stops sending auditory information to the coach client.

5. (say $Message$). Broadcast the $Message$ to all players. $Message$ must be an ASCII string whose length is less than 512.

6. (change_mode $PLAY-MODE$). Change the play-mode to $PLAY-MODE$.

The server also has another option '-coach.w.referee'. Using this option, a coach client can connect the server in which the referee module is still running. In this case, both the referee module and the coach client can control a match.

3.6 Discussion

The soccer server captures enough real-world complexities to be a test bed in a real-time dynamic multi-agent domain. It has following characteristics:
1. It provides multi-agent cooperation, communication and competition among friendly and hostile agents. Communication is limited and not reliable. The message is broadcast to every player by the server. This means that opponents might know the meaning of the message. The player is not sure if its message can be heard or not because, first, the server only accepts one message from each client at every simulation step and chooses the message to be broadcast according to the order of arrival, and, second, because a player can hear only one message from each team during two simulation steps and can only hear messages said by players within a 50 meter range.

2. It provides a real-time dynamic environment. The player’s opponents and teammates change their positions while the player is deliberating.

3. It provides a partially observable environment. The player can only 'see' objects in the 90-degree view cone or within three meters around himself. As the distance to a player becomes larger, more and more information about the player is lost. First, players numbers can not be seen further than a certain distance, then players team names are lost when they are too far away.

4. It provides a nondeterministic environment. The next state of the environment can not be completely determined by the current state and the actions selected by the players because noise is added to each movement according to the speed of the object. The value of distance to the object is also quantized in a certain method so that the noise in the vision information increases when the object moves away from the player.

5. It provides a discrete/continuous hybrid environment. A player can get distinct perceptions, like a referee’s commands, and also the speed and position
of itself and other soccer players sweep through a range of continuous values.
Chapter 4

Designing and Programming the Controller for Soccer-playing Robots

This chapter illustrates the CN based robot architecture and constraint based control by describing how to design and program the controller for soccer-playing robots. Section 1 describes the CN architecture of the controller for a soccer-playing robot. Section 2 discusses constraint-based control for the soccer-playing robot. Section 3 describes the implementation of the controller in Java.

4.1 The CN Architecture of the Controller for a Soccer-playing Robot

The soccer-playing robot system is modeled as an integration of the soccer server and the controller (Fig. 4.1). The soccer server provides 22 soccer-playing softbots'
plants and the ball. Each softbot can be controlled by setting its throttle and steering. When the softbot is near the ball (within 2 meters), it can use the kick command to control the ball's movement. For the controller for one of the soccer-playing softbots, the rest of the players on the field and the ball are considered as its environment. The sensor of the controller determines the state of the plant (position and direction) by inference from a set of landmarks it 'sees'. The rest of the controller computes the desired control inputs (throttle and steering) and sends them to the soccer server to actuate the plant to move around on the field or kick or dribble the ball.

Considering the interaction hierarchy of the CN based architecture, we have designed a three-level controller for the soccer-playing robot shown in Fig. 4.2. The lowest level is the Effector & Sensor. It receives ASCII sensor information from the soccer server then translates it into the World model. It also passes commands from the upper level down to the soccer server. The middle level is the Executor. It
tries to translate the action (goal) which comes from the upper level into a sequence of commands and sends them to the lowest level. The Executor also evaluates the situation and sends it to the top layer (Planner). The highest level is the Planner. It decides which action (goal) to take based on the current situation and it may also consider the next action assuming the current action will be correctly finished on schedule.

As for the composition hierarchy, the robot controller is composed of three CN modules. The Planner module forms the highest level. The Executor module forms the middle level. The Effector & Sensor module forms the low level and itself is composed of two sub modules, they are the Effector module and the Sensor module (Fig. 4.2).

The Sensor is an event-driven module. It has one alarm non-buffered input port for receiving ASCII information from the soccer server. The Sensor module wakes up when new information arrives on its input port. It then processes the data, updates the world model, and sends an event to the Executor. The Sensor goes to sleep when there is no information waiting on its socket.

The Executor is an event-driven module. It has one non-alarm, non-buffered input port for receiving updated world model events from the Sensor, and one alarm non-buffered input port for receiving action events from the Planner.

The Executor module receives the event from the Sensor, then it processes the world model and updates the situation states. These situation states tell the Planner if it can kick the ball, if the ball is in its sight, if it is the nearest player to the ball, if there are obstacles on its way, if it is offside, if the action from the Planner has finished or not, and so on. Any change of situation creates an event and triggers the higher level Planner module.
Figure 4.2: The soccer-playing controller hierarchy
The main part of the Executor executes actions passed down from the Planner. It wakes up when it receives an action event from the Planner module. It produces a sequence of commands which are supposed to achieve goals (actions) when they are performed. Some of these commands are sent to the Effector’s `Movement_command` input port. Other commands are sent to the Effector’s `Sensing_command` input ports, they are `Say_message` input port, `Change_view` input port, and `Sense_body` input port. The Executor goes to sleep when there is no action waiting for its processing.

The Planner is an event-driven module. It has one alarm non-buffered input port for receiving updated situation events from the Executor. The Planner module wakes up when triggered by a situation-changed event from the Executor. It then produces an action event and pushes it into the Executor’s action input port and then it goes to sleep until a new event comes.

The Effector is a fixed-sample-time-driven module. It has one non-alarm hybrid input port for receiving movement commands from the Executor. It is set to hybrid because some movements should be executed in sequence and be treated as a single, indivisible, atomic action. It has three other non-alarm, non-buffered input ports. They are `Sensing_command` input port, `Say_message` input port and `Change_view` input port. Every 100ms, it gets one command from each non-empty port and sends them to the soccer server.
4.2 Constraint-Based Control for Soccer-playing Robot

4.2.1 Constraint Methods in the Executor

The Executor module can be seen as an embedded constraint solver on its world state space. It solves the constraint-based requirements passed down from the higher layer Planner module. There are 7 major actions (constraint-based requirements):

1. *Find its own position.* If the player has lost its own position for a certain period, which is caused by running out of field and not having seen enough landmarks, it turns around until it finds enough landmarks to calculate its own position.

2. *Find the ball.* If the player has not seen the ball for a certain period, it turns around until it finds the ball.

3. *Avoid getting stuck.* If several players run into each other, they get stuck together. When this happens, the player dashes backwards. A sequence of commands act as this atomic action that will be passed down to the effector.

4. *Avoid being off-side.* When the player finds itself at an off-side position, it dashes towards its own side. A sequence of commands act as this atomic action which will be passed down to the effector.

5. *Kick the ball.* The planner passes down the kick action with its parameters telling which direction and position to kick the ball to, and how fast the ball should be at the destination. This is not a trivial problem. We have to consider the dynamic simulation inside the soccer server to set the right kick direction and power. Sometimes the required ball speed can not be reached, then the
Executor will provide the fastest it can. Sometimes even the required direction
can not be set, then the Executor will just kick the ball to the opponents’ side.

6. Intercept the ball. This is also not as trivial a problem as it looks. The simplest
method is to always let the player dash to the ball’s present position. This may
eventually let the player catch the ball. But a better method is for the player
to predict the ball’s future position and dash to there. The best method is
for the player to predict ball-player collision position and dash to it (Fig. 4.3).
In this problem, we know the ball’s position and velocity, we also know how
fast the player can run. We want to know in which direction the player should
dash to intercept the ball. Let \((X_b, Y_b)\) be the ball position, \((V_{bx}, V_{by})\) be the
ball velocity; \((X_p, Y_p)\) be the player position, \((V_{px}, V_{py})\) be the player velocity.
The constraint requirement is that after time \(t\), the player and the ball meet
at a certain position.

So we have the constraint equations:

\[
\begin{align*}
X_p + V_{px}t &= X_b + V_{bx}t \\
Y_p + V_{py}t &= Y_b + V_{by}t
\end{align*}
\]

(4.1)

Let \(V_p\) be the player speed, \(V_b\) be the ball speed; we have

\[
\begin{align*}
V_{px}^2 + V_{py}^2 &= V_p^2 \\
V_{bx}^2 + V_{by}^2 &= V_b^2
\end{align*}
\]

(4.2)

Let \(X_b - X_p = R_x\), \(Y_b - Y_p = R_y\); we have

\[
\begin{align*}
V_{px}t &= R_x + V_{bx}t \\
V_{py}t &= R_y + V_{by}t
\end{align*}
\]

(4.3)
Then we have

\[
\begin{aligned}
V_{px}^2 t^2 &= (R_x + V_{bx} t)^2 \\
V_{py}^2 t^2 &= (R_y + V_{by} t)^2
\end{aligned}
\]  

(4.4)

Then by adding these two equations together, we have

\[
V_p^2 t^2 = R_x^2 + 2R_x V_{bx} t + R_y^2 + 2R_y V_{by} t + V_b^2 t^2
\]

(4.5)

Rearranging, we have

\[
(V_b^2 - V_p^2) t^2 + 2(R_x V_{bx} + R_y V_{by}) t + (R_x^2 + R_y^2) = 0
\]

(4.6)

Now by solving this equation, we can know how long it will take to intercept the ball. Then we can find the collision position because we already know the ball's velocity. When the collision position is decided, the player can turn to it and dash to that position.

From Fig. 4.3, we know there might be two collision positions. Of course, we choose the closer one. In the equation, it means \( t \) has two solutions, we pick the smaller one. Sometimes there is no solution for the equation, then we let the player dash to the position where the ball stops.

In the constraint equations, we suppose the ball and the player are traveling at constant speeds. Actually this is not the case in the soccer server. So according to the dynamic simulations in the soccer server, we made some adjustments to the speeds we used in the equations.
Figure 4.3: The player intercepts the ball, by setting different directions, the player can intercept the ball at two different positions when its speed is constant.

7. *Go to position.* If the destination is \((X_d, Y_d)\), and the player is at \((X_p, Y_p)\), the set of constraints are \(X_d = X_p, Y_d = Y_p\). The player turns to the destination and dashes to it.

To determine the effectiveness of the ball interception method, we equipped one team with our constraint based method, and another team with the method predicting the ball’s position 500 ms into the future.

The results are shown in table 3.1.

<table>
<thead>
<tr>
<th>Game</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constraint based Team</td>
<td>6</td>
<td>1</td>
<td>5</td>
<td>2</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>Normal Team</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

We can see the constraint based team is much better. In six games, it won five games, and only lost one.
4.2.2 Constraint Methods in Planner

The Planner module can be seen as an embedded constraint solver on its situation state space. The ultimate constraint here is: the number of goals should be more than its opponent's. To satisfy this ultimate constraint, the robot has to satisfy a series of other constraints first.

These constraints have their priorities, we call it the arbitration hierarchy. The constraints with higher priority must be solved earlier than those with lower priorities.

The planner chooses actions to satisfy the constraints according to their priorities. Finding its own position has the highest priority. When the robot loses its own position for a certain amount of time, it sends find_me action down to the Executor. Finding the ball is the second priority, when the robot can't find the ball for a certain amount of time, it sends find_ball action down to the Executor. Avoiding collision is the third priority. When the robot senses that it will collide with other players, it sends an avoid_collision action down to the Executor. Avoiding offside is the fourth priority. It sends down an avoid_offside down to the Executor if it finds itself is at an off-side position.

According to the robots' distance to the ball, three different independent groups of constraints form the fifth arbitration level.

If the robot senses its own team players are nearer than itself to the ball, according to the social law constraint, it won't chase the ball, instead it goes to a certain position to cover a certain area on the field. Where it goes depends on its role constraint and the game situation. For example, if it is the goalie, it goes to the front of the goal.

The game has two situations, offensive and defensive. When opponents have
the ball, the situation is *defensive*. When its own team has the ball, the situation is *offensive*. When both teams have the ball, if ball is in the opponents’ side, it is *offensive*, otherwise it is *defensive*.

Each player role has been assigned two different areas to cover. Which one to cover depends on the game situation. If it’s *defensive*, it covers the *defensive* area. When the situation changes to *offensive*, it goes to *offensive* position.

The robot tries to *intercept* the ball if it senses that it is nearer to the ball than its teammates.

When the player is very near the ball, it sends out a message telling its teammates it will control the ball. And its teammates who get the message will give up intercepting the ball if they are doing that and go to certain positions to assist the interceptor.

If the robot is so near the ball so it can kick it, it has to choose where to kick. It can shoot, pass or dribble (pass to itself). Dribble is a dangerous action and is not recommended. Because when the robot is dribbling the ball, the robot and the ball’s speed are so slow that opponent robots can easily kick the ball away.

In order to win, the robot has to consider some constraints, such as, its own team’s time in possession of the ball should be longer than its opponent’s team; the ball should be near enough to the opponent’s goal; the ball should be as far away as possible from its own goal; and the ball should be kicked into opponent’s goal instead of its own goal.

The best action should best satisfy the constraints listed above. Here we have two problems. First, the robot can’t be absolutely sure that certain actions will satisfy certain constraints because the soccer server provides a noisy, dynamic world. For example, it’s impossible for the robot to choose a kick direction which
makes sure that its teammates will get the ball first. We can only say that if the robot choose to kick in this direction, the probability of its teammates getting the ball first is high or low. Secondly, the robot can’t find a kick direction that can maximize all the probabilities of satisfying all the constraints. For example, if the robot chooses the kick direction which makes the probability of its teammates getting the ball very high, the ball might be kicked away from its opponent’s goal and near its own goal.

We solve this by using the combined utility constraint, setting weights for these constraints and combining these constraints into a utility function, which assigns a single number to express the desirability of an action. The Planner chooses the action with the highest utility.

\[ U(a) = \sum_c k_c \cdot P_c(a) \]  

(4.7)

\( U(a) \) is the action \( a \)’s utility. \( P_c(a) \) is the probability of satisfying the constraint \( c \) when taking the action \( a \). \( k_c \) is the weight for the constraint \( i \).

The player can kick in one of 36 directions. These directions are uniformly spaced at 10 degree intervals. The player chooses the kick direction according to the utility function.

The constraints in the utility function are:

1. Keep the ball in the field.
2. Move the ball to the opponent’s side. The probability of satisfying this constraint is high when the kick direction is towards the opponent’s side.
3. Let its own team’s time in possession of the ball be longer than its opponent’s team. We calculate the time each player needs to intercept the ball. If one
of its teammates need a shorter time to intercept the ball than its opponents, the probability of satisfying this constraint is high.

4. Shoot into the opponents' goal. When the kick direction is towards the opponents' goal, the probability of satisfying this constraint depends on the distance between the kicker and the opponents' goal. The nearer to the goal, the higher the chance of success.

5. Avoid shooting into its own goal. The probability of satisfying this constraint depends on the distance between the kicker and its own goal.

We have designed a coach program using an evolutionary algorithm to adjust these weights and other parameters in the controller.

Also, the player will send out a message telling its teammates if it gets the ball. Those teammates who get the message will observe themselves to find out if they are in a good position to get the ball. If they think they are, they will tell the passer to pass the ball to them. If the passer gets one such message, it will give this communication constraint a higher priority and pass the ball to the message sender.

Communication is not only used as a cooperation method, but it is also used to share the world information among teammates. The map information is also included in the messages used to cooperate between teammates. And every 10 steps, the player will send out a message to share its map with its teammates.
4.3 Java Implementation of the Constraint Based Controller

The controller for soccer robot is implemented in Java [Jav98]. The Java model is a fundamentally new way of computing, based on the power of networks and the idea that the same software should run on many different kinds of computers, consumer gadgets, and other devices. It is an object-oriented language which has a very good thread programming environment and provides a very good event mechanism.

Java is a good candidate language for constraint based controller implementation because of these features.

The whole Java soccer system has three parts. Part 1 is a soccer player control system, which has 22 classes, with a total of 5000 lines. Part 2 is soccer coach system, which has 13 classes, with a total of 1500 lines. Part 3 is utility system, shared by both the soccer player and the soccer coach, which has 4 classes, with a total of 500 lines. The whole system has 7000 lines in total.

The soccer coach system will be discussed in the next chapter.

4.3.1 CN modules on Java Threads

When the CPU speed increases become less significant in the future, multi-processor computers will become very widespread. Thread programming is an effective method to utilize these powerful computing resources. Because of the inherent parallelism in CN based robot architecture, these CN modules are all implemented as Java threads. Each CN module can be run concurrently on different processors to improve the speed of the controller. Since these modules are event-driven and fixed-sample-time-driven, Java threads can also improve efficiency on a single CPU too. If no
event arrives, they go to sleep so the CPU can deal with other robots. In such a multi-threaded environment where several different threads may be simultaneously delivering events and/or calling methods and/or processing event objects and/or setting properties, special considerations are needed to make sure that they properly coordinate their behavior, using wait/notify and synchronization mechanisms.

Class Sensor, class Executor, class Planner, class Effector are inherited from Thread class in Java.

Class Buffer is used to implement the port on the CN module. The alarm attribute of the port is realized by Java’s wait/notify mechanism.

4.3.2 Event and Event Listener Classes

Events are one of the core features of the Java 1.1 AWT API, Swing and Java Beans architecture. Conceptually, events are a mechanism for propagating state notifications between a source object and one or more target listener objects. Under the new AWT event model, an event listener object can be registered with an event source. When the event source detects that something interesting has happened it calls an appropriate method in the event listener object.

There are four Event classes in the soccer control system: Class ActionEvent, Class WorldEvent, Class CommandEvent, and Class SituationEvent.

Interfaces like ActionListener, WorldListener, CommandListener, and SituationListener are implemented by CN modules to be registered with event sources.

4.3.3 Transceiver Class

Class Transceiver is used to initialize the UDP connection between the soccer player controller and the soccer server. It also provides methods for the Effector to send
Figure 4.4: The class inheritance hierarchy of visual objects

commands to the soccer server and the Sensor to receive information from the soccer server.

4.3.4 Utility Classes

There are two utility classes:

1. Class $Pvector$. It provides operations on 2D vectors.

2. Class $Util$. It provides some useful methods especially for soccer domain.

4.3.5 Visual Objects Hierarchy

The visual information from the soccer server is complicated. Some objects are fixed, some are moving. Different objects have different attributes and also have some attributes in common. It's better to represent these objects in inheritance hierarchy (Fig. 4.4).
Class *Thing* stores common features about visual objects, such as relative distance and relative direction. Class *Moving* is inherited from *Thing* and adds some more attributes like position and velocity. Class *Line* and class *Flag* are also inherited from *Thing*. Each has some more attributes describing the represented line or flag. Class *Ball* is inherited from *Moving*, with one more attribute describing if the ball is nearby. Class *Player* is also inherited from *Moving*, with more attributes describing team name and player number. Class *Self* is inherited from *Player*, with some more attributes describing stamina, direction, view quality, etc.

### 4.3.6 Robot Class

The `main()` method which starts the whole program is in Class *Robot*. The *Robot* starts *Transceiver* to establish the connection with the soccer server. Then it establishes the connection among the CN modules by registering them with events. Finally it starts four CN module threads.
Chapter 5

Designing and Programming the Coach Using an Evolutionary Algorithm

In this chapter, Section 1 describes the evolutionary algorithm for the soccer team. Section 2 describes how to design the coach using this evolutionary algorithm. Section 3 describes the implementation of the coach in Java. Section 4 presents the results. A discussion about this EA is given at the end of this chapter.

5.1 The Evolutionary Algorithm for the Soccer Team

As mentioned in Chapter 2, a pair of chromosomes decides the behaviors of the soccer player. Three groups of genes are considered:

1. Genes represent the weights in the utility function. They decide what to do when the soccer player gets the ball: to shoot or to pass.
2. Genes represent roles' control areas. They decide where to go when the soccer player does not have the ball. Each role has two strategic positions, offensive and defensive. There are 11 roles in the soccer team. Every soccer player has all the functions of these 11 roles. But when the game begins, each player is assigned a role and this role can not be changed during the game.

3. Genes represent the intercepting strategy. They decide the intercepting range threshold and the relation to other teammate interceptors. If the ball is out of intercepting range threshold, the player gives up chasing and goes to strategic positions. If the other teammate interceptor is nearer to the ball, the player also gives up chasing. How much nearer to the ball the other teammate should be is decided by these genes.

There might be many other important decisive parameters in the controller which could be considered as genes.

The coach maintains a population size of six individuals which contain chromosomes. The evolution procedure goes like this:

1. The players in two teams link to the soccer server one by one. By doing so, each player is given an assigned role.

2. The coach randomly divides the population into two equal-sized groups. One is a male group, the other is a female group.

3. The coach selects the best individual from each group to form a pair of parents. The procedure of selection goes like this:

   (a) The coach sends the genetic information on one individual in the group to the players in one team, then sends the genetic information on another
individual in the same group to the players in the opponent team.

(b) The coach starts the game.

(c) The coach finishes the game after 10 minutes.

(d) The coach records the individual whose genetic information makes the winning team.

(e) Go to step (a), repeat until any individual in the group has played against all other individuals in the same group.

So, in the three-member group, the individual who wins twice is the best one. The second best is the one who wins once. The worst is the one who loses twice. In such situations, the best individual is chosen as one of the parents. But sometimes, every individual in the group wins once, like A defeats B, B defeats C, C defeats A. If so, one is randomly chosen as the parent candidate.

If both groups produce parents, they can have 2 children. These 2 children will replace the worst two individuals in the last generation to form a new generation. If one group produces a parent and the other group produces a parent candidate, they can only have one child. This child will be a substitute for the worst individual in the last generation to form a new generation. But if both groups produce parent candidates, they can't have a child. All the individuals in the last generation will still be alive to form a new generation and to face another competition.

4. The coach lets the parents produce children to form a new generation using mutation and crossover, then goes to step 2 to begin a new generation.

For this evolving procedure, one generation needs six matches to select the parents for the next generation. Each match lasts ten minutes. So one generation
needs one hour to produce the next generation.

5.2 The Design of the Coach

The software architecture of the coach is similar to that of the controller for the soccer player. It has three layers. The lowest layer is the Teacher&Observer. It receives ASCII sensor information from the soccer server to get all the players' and ball's positions. It also passes commands from the upper layer down to the soccer server. The middle level is the Tester. It gets the genetic informations from the upper layer and starts/finishes the game and decides which individual is better. The highest level is Nature. It maintains a population, selects a pair of parents and produces children.

As for the composition hierarchy, the coach is composed of three CN modules. The Nature module forms the highest level. The Tester module forms the middle level. The Teacher&Observer module forms the lowest level and itself is composed of two sub modules, they are the Teacher module and the Observer module (Fig. 5.1).

The Observer is a fixed-sample-time-driven module. It has one non-alarm, non-buffered input port for receiving ASCII information from the soccer server. Every 100 ms, it wakes up to process the new information waiting at its input port. It then processes the data, updates all the players' and the ball's positions, and sends an event containing this information to the Tester.

The Tester is an event-driven module. It has one non-alarm, non-buffered input port for receiving Team events which contain both the teams' genetic information from the Nature, and one alarm non-buffered input port for receiving Info events which contain position information from the Observer.
Figure 5.1: The Architecture of the Coach
The Tester module receives the Team event from the Nature. Then it begins to send the command events down to the Teacher. These commands are:

1. Change mode to "before_kick_off".
2. Move the ball to center.
3. Send genetic information to players.
4. Change mode to "kick_off_1".

When the game is on, the Tester monitors and records how long each team possesses the ball, how long the ball is in each side and the score of each team. It also monitors if the game is stuck. That is, players from both teams get stuck together trying to kick the ball. If this happens, the Tester will stop the game, move the ball to center, and restart the game.

When the game finishes, the Tester will evaluate which team is better and send this Rank event up to the Nature. One team is better than the other if:

1. It scores more.
2. The ball stays longer in the opponents' side if both teams have the same score.
3. It has the ball longer if both teams score the same number of goals and the ball spends an equal amount of time in both sides.

Nature is an event-driven module. It has one alarm non-buffered input port for receiving Rank events from the Tester. Nature divides the population into two equal-sized groups, selects two individuals in a group, sends down the genetic information about these two individuals to the Tester. Then it goes to sleep. It wakes
up when triggered by a Rank event from the Tester which tells which individual is better. In this way, it selects parents, produces children, and forms the next generation.

The Teacher is a fixed-sample-time-driven module. It has one non-alarm input port for receiving commands from the Tester. Every 100ms, it gets one command from the non-empty port and sends them to the soccer server.

5.3 Java Implementation of the Coach

5.3.1 CN modules on Java Threads

Class Nature, class Tester, class Observer, class Teacher are inherited from Thread class in Java.

There are four Event classes in the soccer coach system: Class InfoEvent, Class RankEvent, Class CommandEvent, and Class TeamEvent.

Interfaces like InfoListener, RankListener, CommandListener and RankListener are implemented by CN modules to be registered with event sources.

5.3.2 The Individual and the Population Class

Class Individual is used to represent an individual in EA. It has three arrays to store an individual’s genetic information. Array chromosomeX stores mutated genes from its mother. Array chromosomeY stores mutated genes from its father. Array chromosome is the sum of Array chromosomeX and Array chromosomeY, and can be sent to players as their controllers’ parameters.

Class Population is used to represent the population in EA. It defines the population number and the number of genes stored in an individual. It also defines
the mutation range of each gene. It provides methods to initialize the population and to save the information of a population to a file.

### 5.3.3 Coach Class

The main() method which starts the coach is in Class *Coach*. The *Coach* starts *Transceiver* to establish the connection with the soccer server. Then it establishes the connection among the CN modules by registering them with events. Finally it starts four CN module threads.

### 5.4 Results

After 50 generations of evolution, we pick one team from the 50th generation, and let it compete with the hand-tuned team.

Then after 300 generations of evolution, we let the evolved team compete with the hand-tuned team again.

The results are shown in table 4.1 and 4.2.

Table 5.1: Evolved team from the 50th generation vs. Hand-tuned team

<table>
<thead>
<tr>
<th>Game</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Evolved Team</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Hand-tuned Team</td>
<td>3</td>
<td>4</td>
<td>6</td>
<td>3</td>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 5.2: Evolved team from the 300th generation vs. Hand-tuned team

<table>
<thead>
<tr>
<th>Game</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Evolved Team</td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Hand-tuned Team</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>
An individual in generation 50 still can't beat the hand-tuned opponent. It lost all six games. But an individual in generation 300 performs much better. In six games against the hand-tuned opponent, it won five games and lost one. This shows that EA is successful here.

5.5 Discussion

We set the population size to six, but it would have been better to have a larger population. The selection method is tournament-based, which is time-consuming and population-related. A larger population means the life-span of a generation will be longer than that of a generation in a smaller population. When the population size is six, a generation lasts one hour. If the size becomes eight, we need two hours to produce a generation.

We can also use other tournament methods to select parents. There are many possibilities. For example, dividing the population into more groups.

In our experiments, we set the mutation rate to 0.4. We don’t know if this parameter can affect the results and if it can, how it works. In nature, a high mutation rate sometimes can wipe out a species if its population is not very large. In our soccer domain, the soccer species is always there no matter how high the mutation rate is. The mutation rate might affect the speed of evolution. Some experiments are needed to determine this.
Chapter 6

Related Work

The main part of this chapter describes some existing approaches to the robotic soccer domain. A discussion of the differences and similarities of these approaches is contained in the final section.

6.1 Layered Learning

Because of the inherent complexity of multi-agent systems, machine learning is an interesting and promising area to merge with multi-agent systems. Peter Stone and Manuela Veloso give a method called Layered Learning to control their robot soccer players. Layered Learning is an approach to complex multi-agent domains that involves incorporating low-level learned behaviors into higher-level behaviors. A Neural Network (NN) is used to learn a low-level individual behavior (ball interception), which is then incorporated into a basic collaborative behavior (passing). The collaborative behavior is learned via a Decision Tree (DT). Although the DT predicts whether a player can execute a pass, it gives no indication of the strategic value of doing so. For this level of team strategies, Stone presented Team-Partitioned,
Opaque-Transition Reinforcement Learning (TPOT-RL) to allow a team of robotic soccer players to learn how to cooperate to achieve the goal [SV98b].

6.1.1 Learning a Low-Level Multi-agent Behavior

Peter Stone and Manuela Veloso think that it is futile to try to learn intelligent behaviors straight from primitive actions because of the complexity of the domain. Instead, they think that some useful low-level skills must be learned before moving on to higher level strategies.

The low-level skill that Stone identifies as being most essential is the ability to intercept a moving ball [SV98c]. To equip their players with this ability, they provided the players with a large number of training examples and used a supervised learning technique: Neural Networks (NNs).

For each training example, the shooter kicks the ball directly towards the defender with a fixed power. Due to the noisy environment, the ball does not always move directly towards the defender. The defender gathers training data by acting randomly and recording the results of its actions. It follows a procedure like:

\[(BD = \text{ball's distance}, \ BA = \text{ball's angle}, \ TA = \text{turn angle})\]

- If \(BD > 14\), turn(BA);
- If \(BD \leq 14\), set \(TA = \text{Random Angle between -45 and 45}\);
- Record BD, BA, previous BD and TA;
- \(\text{turn}(BA + TA)\);
- \(\text{dash}()\).
It means that when the ball is outside of a given range, the defender simply watches and faces the ball. Then, once the ball is in range, the defender turns a random angle (within a range) away from the ball and dashes. Although the defender misses most (76%) of the time, after about 750 positive examples, it is said that the defender can learn to perform much better.

In order to automate the training process, a coach program is used. At the end of the trial, the coach resets the positions of both players and the ball for another trial.

The goal of learning is to allow the defender to choose the appropriate turn angle (TA) based upon the BD, BA, and previous BD. After a small amount of experimentation with different NN configurations, they settled on a fully-connected net with four sigmoid hidden units and a learning rate of $10^{-6}$. Under this configurations, the defender is able to save almost all the shots despite the continual noise in the ball's movement.

6.1.2 Using Decision Tree Confidence Factors for a Higher-level Decision

Once young soccer players have learned how to control the ball, they are ready to use their skills to start learning how to make decisions on the field and playing as part of a team [SV98a, SV98d].

Although the execution of a pass in the open field is not difficult given the receiver’s ball-interception skill, it becomes more complicated in the presence of defenders. If in the proper position, a defender(also equipped with the same ball-interception skill) may be able to intercept the ball before it reaches the receiver. Thus, the passer is faced with the task of assessing the likelihood that a pass to a
particular receiver will succeed. When deciding whether or not to make a pass, the passer has many possible features of the scenario at its disposal. They chose to use a learning method that is capable of determining for itself which attributes to use. In particular, they used Decision Trees (DTs).

The key part of gathering training examples was the passer’s recording of the attributes describing the trial. Rather than restricting the number of attributes, they capitalized on the DT’s ability to filter out the irrelevant ones. Thus, they gathered a total of 174 attributes for each trial. The attributes from the receiver’s perspective were communicated to the passer before it had to decide which player to pass to. The attributes—all continuous-available to the DT were:

- Distance and Angle to the receiver(2);
- Distance and Angle to other teammates (up to 9) sorted by angle from the receiver (18);
- Distance and Angle to opponents (up to 11) sorted by angle from the receiver (22);
- Counts of teammates, opponents, and players within given distances and angles of the receivers (45);
- Distance and Angle from receiver to teammates (up to 10) sorted by distance (20);
- Distance and Angle from receiver to opponents (up to 11) sorted by distance (20);
- Counts of teammates, opponents, and players within given distances and angles of the passer from the receiver’s perspective (45).
Whenever fewer than the maximum number of players were visible, the remaining attributes were marked as unknown.

The goal of learning is to use these attributes to predict whether a pass to the given receiver will lead to a SUCCESS, a FAILURE, or a MISS. For training, they used standard off-the-shelf C4.5 code with all of the default parameters.

In order to test the DT's performance they ran 5000 trials with the passer using the DT to choose the receiver. Since the DT returns a confidence estimate in its classification, the passer can choose the best receiver candidate even if more than one is classified as likely to be successful.

Their controller relies on a simple communication protocol between the passer and potential receivers. When a player is going to use the DT to estimate the likelihood of a pass succeeding, it alerts the teammate that the pass is coming, and the teammate, in turn sends some data reflecting its view of the world back to the passer.

They define a function called Receiver Choice Function (RCF), that determines what the robot should do when it has possession of the ball. The DT is incorporated into the Receiver Choice Function.

Each robot is assigned a particular position on the field, or an area to which it goes by default. Unless chasing the ball, the robot goes to its position, moving around randomly within a small range of the position. The robot chases the ball whenever it thinks that he is the closest team member to the ball. Although the robot will not interfere his teammate's action by this method, the problem of their controller is if the player is covered by one opponent, it can't move to open positions on the field.

Using a DT, their players learned to judge the likelihood that a pass to a
given receiver would be successfully received. This judgment represented a second layer in their quest to build intelligent Soccer Server clients by layered learning.

6.1.3 Team-Partitioned, Opaque-Transition Reinforcement Learning for Team-level Strategies

They applied TPOT-RL to enable each teammate to simultaneously learn a high-level policy which is a function that determines what an agent should do when it has possession of the ball.

The predictions made by the DT are used to generate a state space for TPOT-RL, then a Q-learning function is used to estimate the expected reward for taking each possible action. An action is chosen for execution and its real-world reward is used to further update the Q-learning function.

6.2 Co-Evolving Soccer-playing Robot with Genetic Programming

Sean Luke used Genetic Programming (GP) to evolve coordinated team behaviors and actions for the soccer players [LHF+97].

Genetic programming is one kind of Evolutionary Algorithm. The objects that constitute its population are not fixed length character strings that encode possible solutions to the problem, but are programs that, when executed, “are” the candidate solutions to the problem. These programs are expressed as parse trees in genetic programming.

GP’s mutation operator takes a single individual, replaces an arbitrary subtree in this individual with a new, randomly-generated subtree, and adds the new
born individual to the next generation.

GP's crossover operator swaps randomly-generated subtrees among two fit individuals to produce two new individuals for the next generation.

GP's reproduction operator simply takes a fit individual and adds it to the next generation.

First, Luke wrote a set of low-level "basic" behaviors that are so simple that there is little reason to "evolve". He thought "go to ball" is worth evolving because the player must *intercept* a moving ball. Another interesting behavior which is evolved is determining which teammate is the best to kick to, or if a kick to the goal is better than passing.

They divide their team into squads. Each squad develops a separate algorithm used by all the players within the squad. The algorithm for a squad consists of two separate tree-like programs. One executed whenever the player is able to kick the ball, and the other executed when it can see the ball but cannot kick it. Both programs take various information about the state of the world as input, and output a $<\text{Distance}, \text{Direction}>$ vector indicating an action (turning or kicking when in possession of the ball, turning and dashing when not).

To evaluate the fitness of all the teams in the population, it first pairs off teams in the population, then plays matches for each pair. Fitness is based on a variety of factors including the number of goals, time in possession of the ball, and average position of the ball. The resultant fitness assessments are used by the GP system to determine the selection and breeding to form the next generation of soccer teams. From [LHF+97], it is not clear what methods they use to select parents and produce offspring.
6.3 Explicit Teamwork Model

Milind Tambe and his colleagues built their soccer team on a domain-independent explicit teamwork model called STEAM [MAAO+98, Tame, Tama, Tamb].

Their agent uses a two-layered architecture, consisting of a higher-level that makes decisions and a lower-level that handles various time critical functions linked to perception and action.

There are two key aspects of their approach to teamwork. The first is the explicit representation of team activities via the use of explicit representation of team operators. Team operators explicitly express a team’s joint activities.

The second key aspect of teamwork is its approach to coordination and communication via the use of a general-purpose teamwork model. STEAM models team members’ responsibilities and commitments in teamwork in a domain independent fashion. STEAM uses the formal joint intentions framework as its basic building block. It uses decision-theoretic reasoning to select the appropriate method for attaining mutual belief. Thus, it does not rely exclusively on explicit communication.

6.4 Discussion

There are many other approaches to this robotic soccer domain. Some of them emphasize the robotic architecture. Some of them emphasize machine learning. In the machine learning group, some prefer Neural Network, others like Reinforcement Learning. Interested readers are referred to [Asa98, Kit97].

As for the robotic architecture, many approaches use layered architecture. Most of them don’t have a clear defined interactive mechanism between layers. Our CN based architecture is different from them because:
• It is object oriented.

• It is event-flow based.

• It is parallel.

Machine Learning is suitable if the problem is so complex that we do not know its function representation or we know its function but the function is NP hard or solving the function in real-time is impossible.

Some approaches use Neural Network to solve intercepting the ball. We think that an analytic method might be more suitable for this problem. Our constraint based method provides an analytic solution to this problem.

Those who emphasize multi-agent teamwork often treat other agents as special objects apart from the environment. Our approach treat other agents as part of the whole environment and they affect the agent with multi-agent constraints.
Chapter 7

Conclusions and Further Research

7.1 Results and Evaluation

7.1.1 World RoboCup98

To compare our approach with other teams' that are different in models, architectures and control methods, we took part in the World RoboCup98 which was held on July 4-8, 1998 in Paris, France.

The first game we played against NIT Stones 98, which took third place in RoboCup Japan Open 98. The opponent team had an interesting strategy with many of its players swarming around the ball and kicking the ball forward. There was almost no passing. One advantage of this strategy is that off-side seldom happens even if the designer does not consider avoiding off-side. One big disadvantage is that the team can only cover a small area. We won this game, with a score of 4:1.

We played against Mainz Rolling Brains in the second game. This team's
strategy was to move the full-backs up in an offside trap to push the opponents' forwards back. But its forwards didn't try to avoid offside positions, they just kept their positions near the opponent's goal. This strategy was used by many teams in World RoboCup98 and should be the best strategy if there was no offside rule. We drew this game, the score was 0:0. Our team's advantage is that our players can sense if they are at offside positions, and if they are, they can try to avoid that situation by moving towards their own side. Our players' low level skills like kicking backwards were not as good as those of the opponent's team. Lots of shots by the opponents were saved because their forwards were offside. Our players advanced near the opponent's goal many times, but their shots lacked adequate strength to score.

We played against CAT-Finland in the third game. This team's original strategy was to keep its full-backs near its own goal and its forwards near the opponent's goal. It's a fixed position strategy and it was also used by many teams in World RoboCup98. When CAT-Finland competed with Mainz Rolling Brains, the disadvantage of their strategy was shown in the score 0:4. When CAT-Finland played against our team, they changed their strategy to that used by Mainz Rolling Brains. Some teams belonging to this category also changed their strategy later as CAT-Finland did. We lost this game; the score was 0:1. Lots of shots by CAT-Finland were also saved because their forwards were offside. At one point, one of our full-backs slowed down to keep energy, so CAT-Finland's forwards got chance to shoot. Our goalie missed the ball.
7.1.2 Discussion

Our team won one game, drew one game and lost one game in World RoboCup98. Although we lost the game, we don't think our team is worse than CAT-Finland. We know there are many random factors in the soccer server and network communication between the server and clients is not stable either. Winning was not our purpose. Our team was successful in the World RoboCup98 from a research point of view. It shows that constraint-based control and evolutionary algorithms are effective methods in multi-agent real-time robot design. It also shows that Java is fast enough to compete in a traditional C++ world.

7.2 Conclusions

Soccer as a task domain is sufficiently rich to support AI research in a multi-agent, real-time dynamic, partially observable, nondeterministic, discrete/continuous hybrid environment.

In this thesis, we propose a constraint-based approach to robot control in such a complex domain. This approach has two major parts:

- Constraint Net (CN) based Robot Architecture. It supports composition hierarchies which allow a complex module to be incrementally composed of simpler ones. It also uses the idea of interaction hierarchy. The bottom level sends control signals to various effectors, and at the same time, senses the state of effectors. Control signals flow down and the sensing signals flow up. Sensing signals from the environment are distributed over levels. Each level is a black box that represents the causal relationship between the inputs and the outputs. The inputs consist of the control signals from the higher level, the
sensing signals from the environment and the current states from the lower level. The outputs consist of the control signals to the lower level and the current states to the higher level.

This CN architecture is object-oriented, parallel, and event-flow based. Events are used for communication among these CN modules. An event is an object which contains signals and (or) data. There are two types of CN modules in this architecture, *event-driven* or *fixed-sample-time-driven*. An *event-driven* CN module does nothing until it is woken by several events sent by other CN modules. A *fixed-sample-time-driven* CN module works and rests according to a module-dependent fixed time schedule.

CN modules have different types of input ports for receiving events to coordinate their behaviors. According to the effect that wakes up the event-driven CN module, we define two kinds of input ports, they are *alarm* input ports and *non-alarm* input ports. As for buffering, there are three types of input ports, *non-buffered*, *buffered* and *hybrid*.

- Constraint-based control methods. Constraints are specified at different levels on different domains, with the higher levels being more abstract and the lower levels being more plant-dependent. A control system can also be synthesized as a hierarchy of interactive embedded constraint solvers. Each abstraction level solves constraints on its state space and produces the input to the lower level.

Constraint solvers at the same level of the interaction hierarchy are coordinated via various arbitrations to compromise among different kinds of constraint.

Another way to deal with various constraints at the same level of the interac-
tion hierarchy is to combine these constraints into one utility constraint. This combined utility constraint is to maximize the utility function.

\[ U(o) = \sum_i k_i \cdot P_i(o) \]  

(7.1)

\( U(o) \) is the action \( o \)'s utility. \( P_i(o) \) is the probability of satisfying the constraint \( i \) when taking the action \( o \). \( k_i \) is the weight of the constraint \( i \).

An evolutionary algorithm (EA) is used to adjust the weights in the utility function and other parameters in the controller.

Based on the Constraint Nets (CN) model, we simply treat other agents as a part of the environment. This multi-agent part of the environment also makes some constraints on the agent as the rest of the environment does.

This constraint-based approach could be applied to many other single or multi-agent real-time dynamic domains. Of course the users would still have to change the interfaces and constraint-based methods to suit their domains.

7.3 Contributions

The major contributions to robot design and implementation are:

- The event-flow based coordination among CN modules.
- Using combined utility constraints to deal with various constraints at the same level of the interaction hierarchy.
- Using an evolutionary algorithm to tune the performance of the controller.
- Treating multi-agent cooperation and competition as constraints on the agent.
7.4 Further Research and Work

More work is needed on the low level constraint based control on how to manipulate the ball more skillfully, like dribbling and kicking backwards. Stamina control should also be considered carefully. Under normal situations, the player can slow down to maintain energy; but when the situation is critical, the player should perform the task no matter how much energy it has.

As for the utility function, we can try to use Neural Networks to represent it instead of a linear representation. Evolutionary Algorithms can also be used to adjust the weights in Neural Networks. Reinforcement learning is also a possible way to learn this function.

We can also change the selection and breeding methods in evolutionary algorithms to find out if there is any difference. The mutation rate can be changed to demonstrate its effects on the evolutionary algorithm.

We should let a human-controlled soccer team compete with robot-controlled soccer team. That is: 11 human players control 11 players on one team, and 11 robot programs control 11 players on the opponent team. The current soccer server has complicated control commands which should be sent by the clients for every 0.1 second so that a human player could not control his player in real-time as well as the robots. We could simplify the interface between client and the soccer server, by letting the soccer server do the low level actions, so humans could compete with robots to show who are better team players; humans or robots? We shall see.
Appendix A

Inside the Soccer Server

A.1 Format of Visual Information

The format of 'see' information is as follows:

\((\text{ObjName Distance Direction DistChng DirChng})\)

\(\text{Distance, Direction, DistChng and DirChng} \) are calculated in the following way:

\[
\begin{align*}
\text{Prx} &= p_{xt} - p_{x0} \\
\text{Prx} &= p_{yt} - p_{y0} \\
\text{Vrx} &= v_{xt} - p_{x0} \\
\text{Vry} &= v_{yt} - p_{y0}
\end{align*}
\]

(A.1)  
(A.2)  
(A.3)  
(A.4)
\[ \text{Distance} = \sqrt{p_{rx}^2 + p_{ry}^2} \quad (A.5) \]

\[ \text{Direction} = \arctan\left(\frac{p_{ry}}{p_{rx}}\right) - a_0 \quad (A.6) \]

\[ e_{rx} = \frac{p_{rx}}{\text{Distance}} \quad (A.7) \]

\[ e_{ry} = \frac{p_{ry}}{\text{Distance}} \quad (A.8) \]

\[ \text{DistChng} = (v_{rx} * e_{rx}) + (v_{ry} * e_{ry}) \quad (A.9) \]

\[ \text{DirChng} = \left[\left(- (v_{rx} * e_{ry}) + (v_{ry} * e_{rx})\right)/\text{Distance}\right] \ast \left(\frac{180}{\pi}\right) \quad (A.10) \]

where \((p_{xt}, p_{yt})\) is the absolute position of a target object, \((p_{xo}, p_{yo})\) is the absolute position of the sensing player, \((v_{xt}, v_{yt})\) is the absolute velocity of a target object, \((v_{ox}, v_{oy})\) is the absolute velocity of the sensing player, and \(a_0\) is the absolute direction the sensing player is facing. \((p_{rx}, p_{ry})\) and \((v_{rx}, v_{ry})\) are, respectively, the relative position and velocity of the target, and \((e_{rx}, e_{ry})\) is the unit vector that is parallel to the vector of the relative position.

### A.2 Movements of Objects

In each simulation step, the movement of each object is calculated as follows:

\[ \text{speed} = \sqrt{v_x^2 + v_y^2} \quad (A.11) \]
\[ (u_x^{i+1}, u_y^{i+1}) = (v_x^i, v_y^i) + (a_x^i, a_y^i) + (r_x, r_y) \]  
\[ (p_x^{i+1}, p_y^{i+1}) = (p_x^i, p_y^i) + (u_x^{i+1}, u_y^{i+1}) \]  
\[ (v_x^{i+1}, v_y^{i+1}) = \text{decay} \times (u_x^i, u_y^i) \]  
\[ (a_x^{i+1}, a_y^{i+1}) = (0, 0) \]

\[ (p_x^i, p_y^i) \] and \[ (v_x^i, v_y^i) \] are respectively position and velocity of the object in time-step \( t \). \text{decay} is a decay parameter specified by \textbf{ball\_decay} or \textbf{player\_decay}. \( (a_x^i, a_y^i) \) is acceleration of the object, which is derived from the \textit{Power} parameter in \textbf{dash} (in the case that the object is a player) or \textbf{kick} (in the case of a ball) commands as follows:

\[ (a_x^i, a_y^i) = \text{Power} \times (\cos\theta^i, \sin\theta^i) \]

where \( \theta^i \) is the direction of the object at time-step \( t \). In the case of a player, its direction is calculated as the following manner:

\[ \theta^i = \theta^i + \text{Moment} \times (1.0 + \text{rand})/(1.0 + 5 \times \text{speed}) \]

where \textit{Moment} is a parameter of the \textbf{turn} command. In the case of a ball, its direction is given in the following manner:

\[ \theta_{\text{ball}}^i = \theta_{\text{kicker}}^i + \text{Direction} \]
where \( \theta_{\text{ball}} \) and \( \theta_{\text{kicker}} \) are the directions of the ball and the kicking player, respectively, and \( \text{Direction} \) is the second parameter of a kick command. \( r_x \) and \( r_y \) are random numbers whose distributions are uniformly located between the range \([-r_{\text{max}}, r_{\text{max}}]\). \( r_{\text{max}} \) is a parameter that depends on the velocity of the object:

\[
    r_{\text{max}} = \text{rand} \times \text{speed} \tag{A.19}
\]

A.3 Unreliability of Information about Far Objects

In the case that an object in sight is a ball or a player, the distance to the object is quantized in the following method:

\[
    d' = \text{Quantize}(\exp(\text{Quantize}(\log(d), 0.1)), 0.1) \tag{A.20}
\]

where \( d \) and \( d' \) are the exact distance and quantized distance respectively, and

\[
    \text{Quantize}(V, Q) = \text{rint}(V/Q) \times Q \tag{A.21}
\]

This means that players can not know the exact positions of objects very far away. For example, at a distance of about 100.0, the maximum noise is about 10.0, while for distances less than 10.0, the noise is less than 1.0.

In the case of flags and lines, the distance value is quantized in the following manner:

\[
    d' = \text{Quantize}(\exp(\text{Quantize}(\log(d), 0.01)), 0.1) \tag{A.22}
\]
A.4 Off-side Rule

The off-side rule algorithm is as follows:

- When a player of team-X kicks a ball, players of team-X in off-side positions are marked "off-side".

- When a player marked "off-side" tries to kick the ball (when the distance to the ball is shorter than 10 meters), the referee judges "off-side!".

- the "off-side" mark is cleared by each effective kick.

where, the 'off-side position' is:

- in the opponent's side.

- between the ball and the opponent's goal line.

- between the second defending player and the goal line.
Appendix B

Regulations of The Simulation Track in World RoboCup

B.1 The Format of the Competition

The competition is played in two rounds.

In the first round, teams shall be divided into several groups of 4 to 5 teams. The number of groups depends on the total number of teams. The system of play is the round-robin system, each team plays against each of the other teams in the same group. A team earns 3 points when it wins a game, earns 1 point when it draws a game, and earns no points when it loses a game. The two teams coming first and second in each group are qualified for the second round.

The second round is played by a system of elimination (cup system).
B.2 Whole Process of a Match

The duration of the match is about 10 minutes (6000 simulation steps), consisting of two periods with a 5 minutes break at halftime.

1. Each client of each team connects with the server by an **init** command.

2. When all clients are ready to play, the match commissary (a person who invokes the server) starts the match by pressing the *kick-off* button of the server window. Then the first half starts.

3. The first half is about 5 minutes. When the first half finishes, the server suspends the match.

4. Half-time is 5 minutes. During half-time, competitors can change client programs.

5. Before the second half, each client re-connects with the server by a **reconnect** command.

6. When all clients are ready, the commissary starts the second half by pressing the *kick-off* button. Then the second half starts.

7. The second half is also about 5 minutes. After the second half, the server stops the match.

8. If the match is a draw, an extra half starts if the match is in the second round of the competition (cup system). The extra half ends immediately when a team scores a goal (golden-goal style).
B.3 Rules Judged by the Server

Soccer Server controls a match according to the following rules.

- When a ball is in a goal, the referee announces the goal (broadcasts a message to all clients), renews the score, moves the ball to the center mark, and changes the play-mode to kick_off.

- When a ball is out of the field, the referee moves the ball to a proper position (a touch-line, corner or goal-area) and changes the play-mode to kick_in, corner_kick or goal_kick.

- When the play-mode is kick_off, corner_kick or goal_kick, the referee removes defending players from an area in a circle whose center and radius are the ball and 9.15 respectively.

- When the play-mode is kick_off, corner_kick or goal_kick, the referee changes the play-mode to play_on after the ball starts by a kick command.

- The referee suspends a match when the first or the second half finishes. The length of each half is about 5 minutes (3000 simulation-cycles). If the match is a draw after the second half, the match can be extended until another goal (golden-goal style) is scored.

- At the kick off, all players must be on their own side. If players are on the opponent's side, the referee moves them onto their own side randomly. There is a period of 5 seconds for players to go back to their own side after a goal. During the interim, clients can use the move command to move their players to a certain position immediately. After the time expires, players on the opponent side are moved randomly by the server.
B.4 Rules Judged by Humans

Fouls like "obstruction" are difficult to judge automatically. Therefore, the server prepares a way for humans to judge such fouls.

- If a player is too un-gentlemanly or interferes in a match, the match-commissary (an organizer of the match) can suspend a match by clicking on the player in the field window and restarting the match with a free kick for the opposite team.
Appendix C

The Dynamo98 Robotic Soccer Team

C.1 System Requirements

The Dynamo98 robotic soccer team can run on any platform which has Java VM installed. It is recommended to set the JIT option on. For better performance, it is also recommended to set the native threads option on.

The solaris PCs in the LCI lab are ideal machines for the soccer team. I installed JDK1.2 for the solaris PCs under /lci/project/dynamo/soccer/jdk1.2beta3/. This JDK has JIT and native threads option.

To make a match between two robotic teams, 24 processes are run simultaneously on a networked computer system. Because of the real-time requirement of the soccer server, these processes need to be divided into process groups run on several different machines in the network. Normally, the soccer server and monitor run on one machine. Five processes in one team run on one machine, six processes
in the same team run on another machine. The soccer coach runs on a separate machine. So six computers are needed to run the soccer game.

C.2 The Steps to Run the Dynamo98 Soccer Team Without the Coach

1. `rlogin` lcics2, amber, nelson, bud and taybeh (for example).


3. On amber, `cd /lci/project/dynamo/soccer/`, `source` inteljava, `cd dp8`, `s21 lcics2 TeamLname Genes1`. TeamLname is the team name for the left team. Genes1 is the file name of a parameter file which stores parameters used in the controller.

4. On nelson, `cd /lci/project/dynamo/soccer/`, `source` inteljava, `cd dp8`, `s22 lcics2 Teamr_name Genes1`.

5. On bud, `cd /lci/project/dynamo/soccer/`, `source` inteljava, `cd dp8`, `s21 lcics2 Teamr_name Genes2`.

6. On taybeh, `cd /lci/project/dynamo/soccer/`, `source` inteljava, `cd dp8`, `s22 lcics2 Teamr_name Genes2`.

7. When it is half time, users can start the second half by simply clicking the Kick off button on the soccer monitor. If users want to change parameters, they can use `ka` to kill the clients on each machine. Then

   (a) On amber, `r21 lcics2 TeamLname NewGenes1`.

   (b) On nelson, `r22 lcics2 TeamLname NewGenes1`. 

86
8. When the game finishes, click the Quit button on the soccer monitor to quit the game. Use ka on the solaris PCs to kill the clients.

C.3 The Steps to Run the Dynamo98 Soccer Team With the Coach

1. rlogin lcics2, amber, nelson, bud, taybeh and tecate.
3. On amber, cd /lci/project/dynamo/soccer/, source inteljava, cd dp8, s21 lcics2 Teamr_name Genes1.
4. On nelson, cd /lci/project/dynamo/soccer/, source inteljava, cd dp8, s22 lcics2 Teamr_name Genes1.
5. On bud, cd /lci/project/dynamo/soccer/, source inteljava, cd dp8, s21 lcics2 Teamr_name Genes2.
6. On taybeh, cd /lci/project/dynamo/soccer/, source inteljava, cd dp8, s22 lcics2 Teamr_name Genes2.
7. On tecate, cd /lci/project/dynamo/soccer/, source inteljava, cd dp8, c Logfile. Logfile is used to store the competition results.
8. To view the competition, on lcics2, cd /lci/project/dynamo/soccer/sserver-4.19, check.
9. To stop the evolution, use ka on Solaris PCs to kill the clients.
Bibliography


[HB98] Jorg Heitkotter and David Beasley. The hitch-hiker’s guide to evolutionary computation: A list of frequently asked questions (faq)


91


