Extendable Swarm Programming Architecture

A Thesis

in TCC 402

Presented to

The Faculty of the

School of Engineering and Applied Science

University of Virginia

In Partial Fulfillment

of the Requirements for the Degree

Bachelor of Science in Computer Science

By Adam Trost

July 31, 2001

On my honor as a University student, on this assignment I have neither given nor received unauthorized aid as defined by the Honor Guidelines for Papers in TCC Courses.

_____

Approved _____(Technical Advisor)

David Evans

Approved _____(TCC Advisor)

W B. Carlson

Abstract

Computing is beginning to change as programs start to execute over many mobile processors communicating over *ad hoc* networks. Collections of these processors can be described as a "swarm." The behavior of a swarm is categorized as the total behavior of all its individual components but, unlike traditional distributed programming, swarms exist dynamically in unpredictable environments. The major challenges are designing programs for the units with a desired swarm behavior and, on the other side, predicting behavior from the programs running on the units. The soccer simulation competition within the RoboCup 2001 conference is the medium of the swarm research. This conference uses a soccer simulation to focus on cooperation between autonomous agents in dynamic multiagent environments. The simulation league comprises of a server acting as the field, and eleven clients for each team, which act as the players. The field is an unpredictable dynamic environment, while the players are thought of as the cooperative swarm. The research addresses the challenges of swarms by implementing an extendable object-oriented architecture for a RoboCup soccer player.

Testing the ease of adding the centering and dispersing defensive behaviors displays the benefits of the program architecture. The extendable object-oriented design resulted in easily implementing the two behaviors into the swarm program of the RoboCup soccer player. Though RoboCup is only one application of swarm programming, the architecture can be applied to many others. If utilized, the swarm development community could evolve more effectively with endless potential.

Table of Contents

Glossary

ad hoc - contrived purely for the purpose in hand rather than planned.

architecture - the organization and interaction of classes in a program.

class - programmer defined data structure for an object.

client - computer software and/or hardware utilizing a server's resources.

instance - an individual object of a certain class.

low-level actions - the basic action the server can understand, such as dash and kick.

object - a unique instance of a data structure defined according to the template provided by its class.

object-oriented design - see architecture.

server - computer software and/or hardware whose resources are shared by multiple users.

source code - commands written in some formal programming language which can be compiled automatically.

state of the world - the properties of the game communicated by the server.

timestep - one change in time experienced by the server.

Chapter

1

1 Introduction

1.1 Swarm Programming

Computing is beginning to change as programs execute on many mobile processors communicating over *ad hoc* networks. Of the 8 billion computing units that will be deployed worldwide this year, only 150 million are stand-alone computers [Tennenhouse2000]. Technological advancements in microelectromechanical systems and wireless networking are producing small, inexpensive devices with significant computing and communicating capabilities [Evans2000]. Collections of these mobile processors can now be described as a "swarm." There is great interest in creating and understanding the properties of these computational swarms.

Swarms have an inherent complexity causing difficulties in research. The behavior of a swarm is the total behavior of all its individual components, yet, unlike traditional distributed programming, swarms exist dynamically in unpredictable environments. The swarm must be resilient not only to a harsh environment, but also to unreliable units that may be executing incorrectly. Time and memory have been the most limited resources in conventional computing, but swarm programming is most concerned with the abilities to compute, to maintain state, and to communicate with neighbors. The major challenges are thus designing programs for the units with a desired swarm behavior and predicting that behavior from the programs running on the units.

Analysis of these two problems is simplified by confining the swarm to a set number of devices. This will make best use of the limited time available for swarm simulation, analysis, and modification. Also, because swarms usually contain many more components than the number of RoboCup soccer players, the results are only as valid as the scale of the reduced swarm correlates to a much larger swarm.

1.2 Research Effects

Research aiding the understanding swarm programming could lead to a variety of potential applications.

1.2.1 RoboCup Improvement

An initial impact would be the improvement of the RoboCup tournament. The RoboCup competition focuses on cooperation between autonomous agents in dynamic multiagent environments. Future developers of RoboCup agents would be able to use many of the ideas created in this program, thus improving their research. The simulation league would evolve into more realistic soccer play, approaching the goal of the tournament; a team of robots competing with world-class players by 2050 [Stone2000-b].

1.2.2 Exploration

Imagine dropping many small robots from an airplane or a spacecraft over Mars in order to explore terrain below. In such a hostile environment, the swarm of robots would need to be programmed so that they could cooperatively adapt to unpredictable situations while pursuing the larger goal. The robots would have to react appropriately if one were to be destroyed to cover the same area with fewer robots. Instead of encoding the robots' decisions into the individual devices, the programs should be mechanically generated from a formal description of the behavior of the robots as a group.

1.2.3 Search and Rescue

A variation of the exploration application would be a directed search for a particular object. The program could attract more devices to an area as sensor feedback shows an increased probability of nearing

the goal. This application could be used to search for skiers or hikers lost in the mountains or to find the black box after a plane crash.

1.2.4 Electronic Commerce

In the future, investors may have millions of agents acting on their financial behalf. Agents may be forbidden to act independently to prevent unwanted situations, such as having too many high-risk investments at once. The owner of the agents could define a policy that limits and guides the overall behavior of this swarm of agents [Evans2000].

1.2.5 Military Uses

The military could also use a robot swarm to search for and destroy something, such as an enemy weapons base or a missile launch site. Unfortunately, a military could also abuse this technology by using an army of robots to help take over a region. The use of swarm programming could revolutionize war, and result in catastrophic destruction.

1.3 RoboCup 2001

The international research symposium, RoboCup 2001, is the medium of the swarm research discussed in this report. Because this conference focuses on cooperation between autonomous agents in dynamic multiagent environments, it easily lends itself to the research. The simulation league is the best setting of all the competitions held at RoboCup 2001 in which to conduct the research. The simulation league consists of a *server* acting as the field, and eleven *clients* for each team, which act as the players. The simulation has a monitor display that represents the game (figure 1). The ball is the white circle in the middle of the field, and the players are represented by the circles with numbers, the lighter side being their front. The rest is like a regular soccer field.

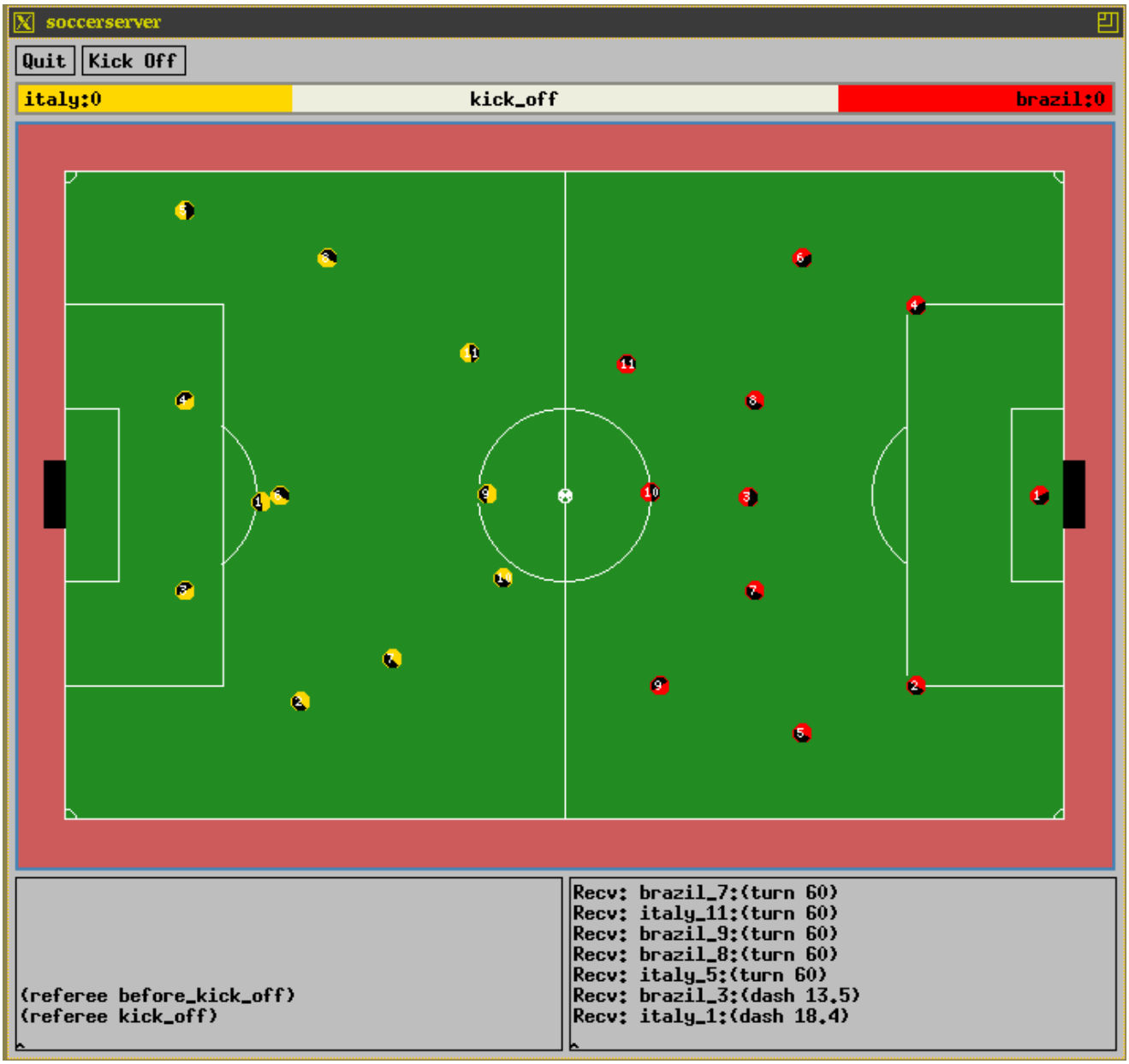

figure 1: The RoboCup Simulation League Display Monitor

The simulation league closely parallels swarms. The field may be considered an unpredictable dynamic environment, while the players are the cooperative swarm. This swarm is more easily analyzed, as there is a relatively small set of units in the team. An aggregate of devised defensive tactics will act as the high-level behavior that the swarm of players will strive to achieve. Players need to maintain constant communication concerning the *state of the world* and of fellow teammates. Defense was chosen because

its basis lies in positioning and overall structure of the entire team, whereas offense concerns more creativity that is individual. The swarm approach should succeed at RoboCup 2001 because its defense will be extremely effective against the opposing offenses, especially with the extensive soccer knowledge of those who created the program.

1.4 Literature Review

The literature research necessary for this thesis pertains to swarm programming and the RoboCup competition. All the swarm programming literature came from a paper written by University of Virginia Professor David Evans and others obtained from a swarm research web site (www.swarm.org). The literature about RoboCup came from the conferences held in previous years. Also, an extremely important document described the *source code* from which the new team is derived.

Evans's paper explains swarm programming, while acknowledging that the field is relatively young so there are many possible approaches. Evans talks about the impact swarm programming could have on society with examples about exploration and sensor networks. Evans' research plan includes creating experimental swarm programs, developing swarm specifications, developing models, analyzing swarm programs, and synthesizing swarm programs. In addition, since Evans is my technical advisor his paper helped set a common vocabulary in which to communicate research ideas. The papers from the swarm web site describe possible tools and techniques for swarm programming. Also, the papers explicitly stated the goals and desired capabilities of programmed swarms. This portion of the literature created a scope for the project, but it lacked specific solutions to the programming problems.

The documentation about the Mainz Rolling Brains RoboCup team was helpful in the early developmental stages. Since the swarm-based team was derived from Mainz Rolling Brains, its document

had to be referenced to more easily understand the source code. Because the document was not comprehensive, at times, the authors had to be contacted via e-mail to answer some questions about their source code. Though the Mainz Rolling Brains team was not created as a swarm program, its adaptation required frequent use of its documentation.

The players in the swarm must make decisions throughout the game. A paper called Mulit-Level Direction of Autonomous Creatures for Real-Time Virtual Environments [Blumberg] was helpful in creating the decision-making method for the RoboCup players. The paper stresses the need for "directability," which is defined as capable autonomous action and response to external control. It uses a robotic dog as an example of how to develop with directability. The paper describes the abstractions of motor skills, sensing, and the behavior system. It also explains the decision-making method of organizing behaviors into mutually inhibiting groups. This allows the dog to make a decision without having two behaviors combine incorrectly. This paper greatly helped the design by offering important ideas about abstraction and behavioral combination.

Finding and utilizing the literature for the RoboCup competition was difficult because most of the papers were not related to this project. The most necessary literature was the documents about creating a team for the competition, including the description of the server functionality so that the program can be compatible. The other papers that were found tended to help brainstorm ideas and approaches to RoboCup, but most were unrelated, concentrated on artificial intelligence.

1.5 Methods

To join the simulation league at the RoboCup competition, entrants must have a program that can interact as a team of clients on the soccer server. This program needs to be developed with not only the

RoboCup rules in mind, but also the concepts of swarm programming. Typically the programs are tens of thousands lines of computer code because programming an intelligent simulation player involves being able to react to myriad situations during the game. This would be nearly impossible to start from scratch with the resources available, so the programming team started working from the C++ source code of the Mainz Rolling Brains team that had previously competed. This code was chosen because, though an overhaul was still necessary, its basic design could be understood and changed more easily than any of the other teams. In addition, a substantial portion of the functionality could be saved with modifications in the code placement and syntax. The changes to the code improved its *object-oriented design*, and abstracted the swarm's basic behaviors away from the rest of the code. This *architecture* allowed for the programmability ideal in swarm programming. I could implement the basic behaviors independently, and then combine them together to choose the action that best helps the swarm achieve its goal. The ease of implementing, testing, and adding the centering and dispersing defensive behaviors displays the benefits of the program architecture. Though RoboCup is only one application of swarm programming, the architecture can be applied to many others. If utilized, the swarm development community could evolve more effectively with endless potential.

Chapter

2

2 Methods

The RoboCup 2001 team created for the simulation league was developed in the C++ programming language. The most important part of the programming for swarm research was the object-oriented design. For effective swarm programming, the architecture abstracts the various components away from each other, such as the state of the world and the definitions of *low-level actions*. When done correctly, extending the program with new behaviors, decisions, and actions is done with less complexity, as discussed in the Chapter 3. (All references to the program *class* architecture in the chapter can be seen visually in figure 2, the program flow.)

2.1 Functionality Abstraction

The architecture abstracts the basic functionality of the program into the three basic layers a RoboCup player experiences.

2.1.1 Listen

Listening is the reading of sensory information by the soccer player's brain. Literally, it is the client processing the data about the state of the world from the soccer server. The Body class uses the Comm class to communicate with the soccer server. Once read, the Comm class packages the information so that it can be passed into the world model by the Body, allowing the player to understand the state of the world.

2.1.2 Think

In this program, thinking is interpreting the world and deciding what action should be taken. Thinking must be correctly implemented if the swarm program is to operate well. The Perception class processes the information in the world model, such as who is the closest opponent to me, for use in behaviors. The behavior classes use the perceptions to choose the action that maintains or initiates that behavior. In the last step of thinking, each behavior passes a recommendation, which is analyzed into the a final action for the player.

2.1.3 Act

Acting is the most straightforward portion of the program. The final action that has been chosen is given back to the Body. The Comm within the Body translates the action so that the soccer server can understand it.

2.2 Class Abstraction

This section will describe the class architecture implemented in the program.

2.2.1 Comm

The Comm class is a two-way interface between the program and the soccer server. Comm understands the protocol and information format that the server gives and receives. Comm parses the information from the server into packages called sensor data. There are different types of sensor data depending on which part of the world model is updated, including the ball, opponents, and teammates. Going the other way, the Comm class takes the final action the player wants to execute and translates it into a message derived from the predefined set of commands the soccer server understands.

2.2.2 Body

The Body class makes communication to the server invisible to the rest of the program. It has a Comm *object* and acts as its interface to the program. The Body sends the sensor data from the Comm

object to the world model, so that it is current. On the other side, the Body takes the final action the player wants and gives it to the Comm object to be communicated to the server.

2.2.3 World Model

The World Model class holds all the basic information the player understands about the state of the world. It uses different types of sensor data to update the corresponding part of the world model. Therefore, at every *timestep*, the world model only updates the portion the player has seen or heard at that time. The player knows that not all the world model is reliable, as aging and indefinite information cannot be fully trusted.

2.2.4 Perception

Perception classes allow the player to process the basic information in the world model usefully. For example, the world model holds the information about the ball and the other players, but a function in a perception would predict which player could get to the ball the quickest. This manipulation of information allows the player to make intelligent decisions about the most effective action.

2.2.5 Behavior

The abstraction of the Behavior classes is the most important in the architecture for effective swarm programming. It allows the program to contain a set of desirable behaviors the swarm analyzes every timestep in the game. Given a circumstance in the world, a behavior uses perception classes to compute the importance of the behavior and the action that best achieves it.

2.2.6 Recommendation

An *instance* of the Recommendation class is produced by each behavior analyzed in a timestep of the game. A recommendation consists of a grade and an action, both set by the corresponding behavior. The purpose of the recommendation is to simplify decision-making. All the recommendations from the behaviors are compared to decide on the final action the player takes.

2.2.7 Action

The Action classes are more intuitive than those communicated to the server. They, along with the perceptions, allow the behaviors to think at a higher level. For example, the player might use the DribbleAction class to dribble the ball without reducing this action to a series of dash and kick commands to the server.

2.3 Decision Making

The C++ object-oriented design uses an algorithm to make the final decision on an action.

2.3.1 Disperse Behavior

The disperse behavior spreads the team defense. Two defenders in the same spot of the field reduces the total area that they can cover. The disperse behavior uses a predefined distance to separate the players. This spreads the defense, but leaves it tight enough to prevent easy passing and dribbling by the offense. The player uses a perception function to find all teammates within the predefined distance. The vectors to those teammates are combined to determine the direction to move to achieve the disperse behavior. The player also gives the action a grade, directly proportional to the proximity of the teammates. The action and grade are set in a recommendation passed on by the behavior.

2.3.2 Central Behavior

The central behavior ensures that the defense protects the team from attack from the center parallel to the lengthwise sidelines. In this part of the field, the offensive players have more dribbling and passing options, and the goal is in this central position. The center of the field perceived by the defense shifts along with the position of the ball, with limits at the edges of the penalty area. For example, if the ball is on the right sideline, the defense should be denser on the right side of the field, but not directly on the right sideline. The behavior will pass a recommendation consisting of the directional move and a grade directly proportional to the distance away from the perceived center.

2.3.3 Final Action Determination

Once all the behaviors have passed recommendations, the player chooses the final action. This process is simple if one recommendation has a much higher grade than the others. For example, if the central behavior's recommendation has a much higher grade than that of the disperse, the player will move towards the center of the field. The decision becomes more difficult with similarly high grades, wherein two possibilities arise. If the behaviors cannot logically combine, then the recommendation with the slightly higher grade is chosen. If the behaviors can logically combine, then the separate actions of those recommendations will aggregate into a final action. For example, if the central and disperse behaviors have high graded recommendations, the player combines them to move in a central direction that disperses from teammates.

Chapter

3

3 Results

The descriptions in this chapter of the swarm programmability show the results of the program architecture. Additions, deletions, and modifications throughout the source code demonstrate the ease at which swarm programming extensions are made with this architecture. In addition, implementing the centering and dispersing behaviors displays a microcosm of the full development, showing how well the object-oriented design lends itself to swarm programming.

3.1 Program Modification

3.1.1 Comm

The Comm object acts as an interface to the server. Thus, when there are version changes to the server, only the Comm class must be changed. If there is a format change, it can take the information and translate it into the same form for the world model. If the server gives a new type of information, then the world model would need to adapt its storage. If a command format is changed, then the Comm simply translates the actions differently. The program can easily be changed to command different forms of players (such as robots) by adapting the Comm translations of the actions.

3.1.2 Actions

The definitions of actions in the program can be added, deleted, or modified with minimal changes to other code. All the behaviors can utilize added actions when deciding the player's next move. When an

action is modified, the programmer must make necessary changes so that behaviors still use the action properly. If an action is deleted, behaviors using it must be modified to use the existing actions without losing desired functionality.

3.1.3 Perceptions

Perception addition and deletion is simple because it corresponds to behaviors. For example, the centering behavior will have a centering perception. When the centering behavior was added, so was the centering perception, and if the centering behavior was deleted, the centering perception would also be deleted. The maintenance of perceptions is the most difficult because of the organization of their functions. If only one behavior uses a perception function, it is put in the corresponding perception. If the function is used by another behavior, the function should move to the perception base class so both perceptions can use it without code duplication. When deleting a perception, the programmer must know all the functions used in the behavior. The functions used only in that behavior are deleted along with the perception, but reorganization must be done for shared functions. If multiple behaviors still use the function, it stays in the perception base class. If only one other perception uses the function, then it moves from the base class to the perception of the behavior that still uses it. If organized correctly, the code length is minimized.

3.1.4 Behaviors

Modification of the behavioral portion of the program is easy. Once initially coded, the behavior goes into the array of behaviors in the master control class. This results in the program analyzing the new behavior along with the others within the think function. The only other change is programming its recommendation to combine with those of the other behaviors. When complete, the programmer can observe how well the new behavior is affecting the swarm, editing until it works as desired. To delete a behavior, the programmer must take it out of the array so that it is not analyzed, and delete its combination with the other recommendations.

3.2 Architecture Programmability

The architecture of this program lends itself extremely well to swarm development. The program's abstraction results in ease of code extension and modification. Though the complete design is not compatible with all forms of swarms, much of it is applicable, especially the behavioral structure. There are some maintenance burdens, but they do not outweigh the benefits. The architecture is exemplified by the development of the centering and disperse behaviors.

The first step to adding a new behavior to the program is to get its functionality working independently. This ensures that its program inclusion is ready, so any problems are isolated in the recommendation combination logic. An initial implementation of the centering behavior was added to the program, but instead of giving its recommendation the computed grade, it is given the highest possible. Therefore, when running the simulation, the player always chooses the centering behavior's recommendation as the final action. This allows the programmer to test many situations, making corrections to the code until the player is moving to the correct place on the field. Once done, the central behavior's recommendation grade was decreased to work on another behavior. The disperse behavior was then programmed similarly. First, the initial implementation and highly graded recommendation were added to the program. Corrections were then made until the players effectively distanced themselves from their teammates.

The central and disperse behaviors were then ready to be combined to achieve correct swarm interaction. Instead of giving the recommendations a predetermined grade, they compute from the state of the world. If the grades are similar, then the recommendation movements combine to achieve a degree of both behaviors. When the two behaviors combined, they did not interact correctly, as the players stayed dispersed without moving centrally. The solution was to weight the multipliers in the centering behavior

equation heavier to improve its grade so that its recommendation would be chosen and combined more often. The two grade equations were adjusted until the players kept central and relatively isolated positions.

Though the programming description only includes two behaviors, it displays the ease of the swarm development. It was a microcosm of the development of many behaviors for a swarm, from independent behavior creation to combination logic. The addition of more behaviors keeps independent behavior development time constant, only increasing the combination complexity. Programmers must make a concerted effort to correctly organize the functions within the perceptions, or the design becomes overloaded and difficult to understand. Reliance on programmers to follow the guidelines of perception organization is an unattractive compromise, but necessary to obtain the desired abstraction. There were few time bottlenecks in the design, leaving only those inherent in the behaviors. Though there are many other development approaches, the programmability of this architecture epitomizes sound design by maximizing swarm programming efficiency and effectiveness.

3.3 Architecture Extensions

Though this design has proven effective for swarm programming, improvements are possible.

3.3.1 Recommendation

The Recommendation class could improve so that the players make decisions that are more intelligent. The major shortcoming of the recommendations is that they only have a single integer grade. Though there are many numbers to vary the grade, little information is passed. A degree of risk and chance for success along with the overall grade may be added so that the players could make decisions such as taking more chances at the end of the game. The change could also require the decision-making architecture to improve by adapting to the more descriptive recommendations.

3.3.2 Actions

There are only basic actions available, such as running, turning, and passing, but there is room to expand. Actions should eventually include multi-stepped moves. This would allow the programmer to execute complex moves more effective against the opposition. An example is a give-and-go pass. Its implementation would require a pass, run, and trap along with communication for teammate coordination of the return pass.

3.3.3 Perception Organization

The organization of perception functions is the most difficult for programmers to maintain in this architecture. It would greatly reduce the unproductive responsibilities of the programmers if the organization was easier if not hidden completely.

Bibliography

[Abelson2000] Harold Abelson, Don Allen, Daniel Coore, Chris Hanson, George Homsy, Thomas F. Knight, Jr., Radhika Nagpal, Erik Rauch, Gerald Jay Sussman, and Ron Weiss. *Amorphous Computing*. Communications of the ACM, Volume 43, Number 5, p. 74-83. May 2000.

[Blumberg] Bruce M. Blumberg and Tinsley A. Galyean. *Multi-Level Direction of Autonomous Creatures forReal-Time Virtual Environments.* Cambridge, MA: MIT Media Lab.

[Catlin1990] Mark G. Catlin, M.D. *The Art of Soccer.* St. Paul, Minnesota: Soccer Books, 1990.

[Chen2001] Mao Chen. *RoboCup Soccer Server.* User's manual for the simulation league for RoboCup 2001, June 2001.

[Daniels2000] Marcus Daniels. *An open framework for agent-based modeling.* Sponsored by Los Alamos National Laboratory and US Marine Corps Combat Development Command, April 2000.

[Evans2000] David Evans. *Programming the Swarm*. Project Proposal Document. July 2000.

[Flenrge2001] F. Flenrge. *Enhancing the Adaptive Abilities Mainz Rolling Brains 2001.* Mainz Rolling Brains RoboCup Simulation League team description document, 2001.

[Heintz2000] Fredrik Heintz. *RoboSoc: a system for Developing RoboCup Agents for Educational Use.* Paper describing the RoboSoc package available at http://www.ida.liu.se/~frehe/RoboCup/RoboSoc/, March 2000.

[Kitano2000] Hiroaki Kitano, Enrico Pagello, and Manuela Veloso, eds. *RoboCup-99: Robot Soccer Wrold Cup 3*. Springer, Germany: Springer, 2000.

[Stone2000-a] Peter Stone and David McAllester. *An Architecture for Action Selection in Robotic Soccer*. Paper submitted to the Fifth International Conference on Autonomous Agents, October 2000.

[Stone2000-b] Peter Stone, ed. *RoboCup-2000: The Fourth Robotic Soccer World Championships.* Paper summarizes the advancements seen at RoboCup-2000, November 2000.

[Stone2001] Peter Stone and Richard S. Sutton. *Scaling Reinforcement Learning toward RoboCup Soccer.* Paper describes the challenges to reinforcement learning in RoboCup agents, 2001.

[Tennenhouse2000] David Tennenhouse. *Embedding the Internet: Proactive Computing.* Communications of the ACM. Volume 43, Issue 5. P. 43-50. May 2000.

[Veloso1998] Manuela Veloso, Peter Stone, and Michael Bowling. *Aniticipation: A Key for Collaboration in a Team of Agents.* Paper submitted to the Third International Conference on Autonomous Agents, October 1998.

Appendix A – Source Code

MASTERCONTROL.H

```
#ifndef _MASTERCONTROL_H_
#define _MASTERCONTROL_H_

#include "recommendation.h"
#include "action.h"
#include "worldmodel.h"
#include "behavior.h"
#include "perception.h"
#include "body.h"
#include "identitydata.h"
#include <iostream.h>
#include <strstream.h>
#include <stdlib.h>
#include <vector>

class MasterControl
{

public:
   MasterControl(char* _pcTeamname, const char* _pcHostName, int _iPort, bool _fLogging,
            int _iReconnectNumber, bool _fIsGoalie = false);
          ~MasterControl();
          void play();
          void listen();
          Recommendation think();
          void act(const Recommendation &_recFinal);

private:
          WorldModel wmWorld;

          vector<Behavior*> pbeBehaviors;
          Body bdBody;
          // filestream for logging
          ofstream out;
          bool fLogging;

   int iLastKickTime;
   bool fClockwise;

   // For logging
   int iLastTime;
   play_mode pmLastPlayMode;

};

#endif
```

```
#include "mastercontrol.h"
#include <cstring>
#include "recommendationlist.h"
#include "centerbehavior.h"
#include "dispersebehavior.h"

using namespace std;

MasterControl::MasterControl(char* _pcTeamname, const char* _pcHostName, int _iPort, bool _fLogging,
                        int _iReconnectNumber, bool _fIsGoalie)
{
        iLastKickTime = UNKNOWN;
        fClockwise = false;

   iLastTime = -1;
   pmLastPlayMode = PLAY_MODE_UNKNOWN;

        fLogging = _fLogging;
        // prepare the identity data and register with the server
        IdentityData idAboutMe;

        idAboutMe.team_name = new char[strlen(_pcTeamname)];
        strcpy(idAboutMe.team_name, _pcTeamname);

        idAboutMe.type = CLIENT_FIELDPLAYER;

        // host information passed in from above, either from command line or as defaults
        bdBody.init(idAboutMe, _pcHostName, _iPort, _iReconnectNumber);

        // insert the IdentityData into the WorldModel
        wmWorld += idAboutMe;

        // initialize behaviors and perceptions
        pbeBehaviors.push_back(new DisperseBehavior(wmWorld));
        pbeBehaviors.push_back(new CenterBehavior(wmWorld));
}

MasterControl::~MasterControl()
{

        unsigned int i;
        for(i = 0; i < pbeBehaviors.size(); i++)
        {
                if (pbeBehaviors[i] != NULL)
                        delete pbeBehaviors[i];
        }
}

void MasterControl::play()
{
        if(fLogging)
        {
        // open log for writing
        char cFilename[80];
```

```
        snprintf(cFilename, 80, "%s%s%d%s", WORLD_LOG_FILENAME, wmWorld.getTeamName(),
         wmWorld.getClientPlayer().getNumber(), WORLD_LOG_EXTENSION);
        out.open(cFilename, ios::out);

        out << "<?xml version=\"1.0\"?>" << endl;
    out << "<!DOCTYPE playerlog SYSTEM \"playerlog.dtd\">" << endl;

    out << "<playerlog>" << endl;
  }

        Recommendation recFinal;

        // Telling the client player move to an initial position
        MoveAction maMove(KICKOFF_POSITIONS[wmWorld.getClientPlayer().getNumber()-1]);
        bdBody.execute(&maMove);

        // while the client's body is connected to the server
        while(bdBody.isConnected())
        {
                // listen
                listen();

                // if the worldmodel has changed
                if ((wmWorld.getTime() != iLastTime) || (wmWorld.getPlayMode() != PLAY_MODE_PLAY_ON))
                {

                iLastTime = wmWorld.getTime();
                        // log the world
                        if(fLogging)
                           out << wmWorld;

                        // think
                        recFinal = think();
                        // act
                        act(recFinal);
                }
        }

        // close log
        if(fLogging)
        {
        out << "</playerlog>" << endl;
        out.close();
  }
}

void MasterControl::listen()
{
        // get information about the world
        bdBody.senseWorld();
        SensorData* psdData;

        while (bdBody.hasSensorData())
        {
                // grab the next piece of sensordata
```

```
                psdData = bdBody.getNextSensorData();
                // insert the new sensordata into the world
                wmWorld += *psdData;

                delete psdData;
                psdData = NULL;
        }
}

Recommendation MasterControl::think()
{
        RecommendationList rclRecommendations;
        Recommendation recTemp;

        unsigned int i;
        // for each behavior
        for(i = 0; i < pbeBehaviors.size(); i++)
        {
                // if the behavior is useful
                if ((pbeBehaviors[i] != NULL) && pbeBehaviors[i]->isUseful())
                        // add it's recommendation to the list
                        pbeBehaviors[i]->evaluate(rclRecommendations);
        }

        // Log the recommendations
        if(fLogging)
        {
           if ((wmWorld.getTime() != iLastTime) || (wmWorld.getPlayMode() != PLAY_MODE_PLAY_ON))
          {
              out << rclRecommendations << endl;
              pmLastPlayMode = wmWorld.getPlayMode();
           }
        }

        // return the composition of all of the recommendations
        return rclRecommendations.compose();
}

void MasterControl::act(const Recommendation &_recFinal)
{
        // get the list of actions from the recommendation
        ActionList acActions = _recFinal.getActionList();
        bdBody.execute(acActions);
}
```

CENTERBEHAVIOR.H

```
#ifndef _CENTERBEHAVIOR_H_
#define _CENTERBEHAVIOR_H_

#include "behavior.h"
#include "centerperception.h"
#include "centerrecommendation.h"
#include "runaction.h"

class CenterBehavior : public Behavior
{
public:
        CenterBehavior(const WorldModel &_wmWorld);
        virtual ~CenterBehavior();

        bool isUseful();
        virtual void evaluate(RecommendationList& recList);

private:
         CenterPerception perCenter;

};

#endif
```

```
#include "centerbehavior.h"

CenterBehavior::CenterBehavior(const WorldModel &_wmWorld): perCenter(_wmWorld)
{
}

CenterBehavior::~CenterBehavior()
{
}

bool CenterBehavior::isUseful()
{
	if (!perCenter.isActiveMode())
		return false;

	bool f = false;
	bool g = false;

	if ( (perCenter.getPossession() != PS_OURS) || (perCenter.inOurPenaltyArea (perCenter.getBall())) )
		f = true;

	if ( (!perCenter.amIClosestToBall()) && (perCenter.getBallAge() < 15) );
		g = true;

	if ( f && g )
		return true;

	else return false;

}

void CenterBehavior::evaluate(RecommendationList& recList)
{
	//create the Recommendation to be passed back
	CenterRecommendation rCenterRec;
	rCenterRec.szBehaviorName = "center";
	//sets the ball position aa the virtual center
	double dVirtCenter = ( perCenter.getBallAbsPos() ).getY();

	//if outside the penalty area width, the respective edge
	//becomes the virtual center
	if ( fabs(dVirtCenter) > 0.5 * PENALTY_AREA_WIDTH - 10.0 )
		if ( dVirtCenter > 0.0 )
			dVirtCenter = 0.5 * PENALTY_AREA_WIDTH - 10.0;
		else
			dVirtCenter = -0.5 * PENALTY_AREA_WIDTH + 10.0;


	//find the maximum distance a player can be away from the virtual center
	double dMaxDist = (0.5 * FIELD_WIDTH) + (0.5 * PENALTY_AREA_WIDTH - 5.0);

	//find fraction to virtual center and get initial grade
	double dDistToVCenter = perCenter.distToVirtCenter ( dVirtCenter  );
```

```
        double dPercToVCenter = fabs (dDistToVCenter) / dMaxDist;
        int iGrade = static_cast<int> (dPercToVCenter * CENTERING_MULTIPLIER);

        //add the bonus depending on where the ball is and set grade
        if (perCenter.inAttackingThird (perCenter.getBall()))
                iGrade += ATTACKING_BONUS;
        else if (perCenter.inMiddleThird (perCenter.getBall()))
                iGrade += MIDDLE_BONUS;
        else if (perCenter.inOurPenaltyArea (perCenter.getBall()))
                iGrade += PENALTY_AREA_BONUS;
        else if (perCenter.inDefensiveThird (perCenter.getBall()))
                iGrade += DEFENSIVE_BONUS;

        rCenterRec.setGrade(iGrade);

        //set the action of the centering move and return recommendation
        Angle aRunAngle;
        if (dDistToVCenter > 0.0)
                {
                aRunAngle = - PI_2 - perCenter.getBodyDir();
                }
        else
                {
                aRunAngle = PI_2 - perCenter.getBodyDir();
                }

                RunAction raTemp;
                pmaTemp->setIntensity(iGrade);
                if (perCenter.inDefensiveThird (perCenter.getBall()))
                raTemp.setDirection(aRunAngle);

        rCenterRec.addAction ( raTemp );

   recList.insert(rCenterRec);
}
```

CENTERPERCEPTION.H

```
#ifndef _CENTERPERCEPTION_H_
#define _CENTERPERCEPTION_H_

#include "perception.h"
#include "types.h"
#include "constants.h"

class CenterPerception : public Perception
{
public:
        CenterPerception(const WorldModel &_wmWorld);
        virtual ~CenterPerception();

        Vector2d getBallAbsPos();
        bool inAttackingThird(WorldObject _woObj);
        bool inMiddleThird(WorldObject _woObj);
        bool inDefensiveThird(WorldObject _woObj);
        bool inOurPenaltyArea(WorldObject _woObj);
         WorldBall getBall();

        double distToVirtCenter (double _dVirtCenter);

        Angle getBodyDir();

        bool isGoalie() const { return wmWorld.getClientPlayer().isGoalie(); }
};

#endif
```

C E N T E R P E R C E P T I O N . C P P

```
#include "centerperception.h"

CenterPerception::CenterPerception(const WorldModel &_wmWorld) : Perception(_wmWorld)
{
}

CenterPerception::~CenterPerception()
{
}

Vector2d CenterPerception::getBallAbsPos()
{
//returns the absolute position of the ball
return wmWorld.getWorldBall().getAbsPosition();
}

bool CenterPerception::inAttackingThird(WorldObject _woObj)
{
	//gets the absolute position of the WMO
	Vector2d vAbsPos = _woObj.getAbsPosition();

	//finds if the X coord is greater than 1/6 Length because greater than that
	//is the attacking third of the field
	if ( vAbsPos.getX() > (FIELD_LENGTH / 6.0) )
		return true;
	else return false;}

bool CenterPerception::inMiddleThird(WorldObject _woObj)
{
	//gets the absolute position of the WMO
	Vector2d vAbsPos = _woObj.getAbsPosition();

	//finds if the X coord is less than 1/6 Length and greater than -1/6 Length
	//because that is the middle third of the field
	if ( (vAbsPos.getX() < (FIELD_LENGTH / 6.0)) && (vAbsPos.getX() > -(FIELD_LENGTH / 6.0)) )
		return true;
	else return false;}

bool CenterPerception::inDefensiveThird(WorldObject _woObj)
{
	//gets the absolute position of the WMO
	Vector2d vAbsPos = _woObj.getAbsPosition();

	//finds if the X coord is less than -1/6 Length because less than that
	//is the defensive third of the field
	if ( vAbsPos.getX() < -(FIELD_LENGTH / 6.0) )
		return true;
	else return false;
}

bool CenterPerception::inOurPenaltyArea(WorldObject _woObj)
{
  Vector2d vAbsPos = _woObj.getAbsPosition();
```

```
        return ( ( vAbsPos.getX() > - 0.5 * FIELD_LENGTH ) &&
          ( vAbsPos.getX() < - 0.5 * FIELD_LENGTH + PENALTY_AREA_LENGTH ) &&
          ( fabs( vAbsPos.getY() ) < 0.5 * PENALTY_AREA_WIDTH ) );
}

WorldBall CenterPerception::getBall()
{
        //returns the world ball
        return wmWorld.getWorldBall();
}


double CenterPerception::distToVirtCenter (double _dVirtCenter)
{
                return  (wmWorld.getClientPlayer().getAbsPosition().getY() - _dVirtCenter);
}

Angle CenterPerception::getBodyDir()
{
        return ( wmWorld.getClientPlayer().getBodyDir() );
}
```

DISPERSEBEHAVIOR.H

```
#ifndef DISPERSEBEHAVIOR_H
#define DISPERSEBEHAVIOR_H

#include "behavior.h"
#include "disperseperception.h"

class DisperseBehavior : public Behavior  {
public:
	DisperseBehavior(const WorldModel &_wmWorld);
	virtual ~DisperseBehavior();

	bool isUseful();
	virtual void evaluate(RecommendationList& recList);

private:

	DispersePerception perDisperse;

};

#endif
```

```
#include "dispersebehavior.h"
#include "runaction.h"

DisperseBehavior::DisperseBehavior(const WorldModel &_wmWorld): perDisperse(_wmWorld)
{
}

DisperseBehavior::~DisperseBehavior()
{
}

bool DisperseBehavior::isUseful()
{
	if (!perDisperse.isActiveMode())
		return false;

	//behavior is useful for offense and defense
	if ((perDisperse.getMyX() > - 0.5 * FIELD_LENGTH) && (!perDisperse.amIClosestToBall()) &&
	 (perDisperse.getBallAge() < 15) )
		return true;

	return false;
}

void DisperseBehavior::evaluate(RecommendationList& recList)
{


	//create the Recommendation to be passed back
	Recommendation rDisperseRec;
	rDisperseRec.szBehaviorName = "disperse";

	//figure out what the disperse distance should be
	double dDispDist;
	if (perDisperse.getPossession() == PS_OURS)
		dDispDist = OFFENSIVE_DISPERSE_DIST;
	else
	{
		double dFieldPerc =  ( (0.5 * FIELD_LENGTH) + (perDisperse.getMyX()) ) / FIELD_LENGTH;
		dDispDist = dFieldPerc * DEFENSIVE_DISPERSE_MULT;
	}

	if (dDispDist <  5.0 )
		dDispDist = 5.0;

	//figure out a vector with the positions of the players within disperse distance

	Vector2d vDisperseVec (0.0, 0.0);
	Vector2d vDist;
	double dDistMagnitude;

	for (int i = 0; i < NUM_TEAMMATES; ++i)
	{
```

```
                vDist = perDisperse.relPos(perDisperse.ourTeamDistIndex(i));
                if ((vDist.getLength() < dDispDist) && (vDist.getLength() != 0.0))
                {
                        dDistMagnitude = vDist.getLength();
                        vDisperseVec += ((vDist / dDistMagnitude) * (dDispDist - dDistMagnitude));
                }
        }

        // take the negative of the composite distance vector and set recommendation
        vDisperseVec = -vDisperseVec;

        perDisperse.checkVec(vDisperseVec);

        double dGrade = (vDisperseVec.getLength()  / dDispDist) * 100;


        rDisperseRec.setGrade(dGrade);

        Angle aDir = vDisperseVec.getAngle();

 RunAction raTemp;
 raTemp.setDirection(aDir);
 pmaTemp->setIntensity(iGrade);
 rDisperseRec.addAction(raTemp);

        recList.insert(rDisperseRec);
        return;

}
```

DISPERSEPERCEPTION.H

```
#ifndef DISPERSEPERCEPTION_H
#define DISPERSEPERCEPTION_H

#include <perception.h>

class DispersePerception : public Perception  {
public:
        DispersePerception(const WorldModel &_wmWorld);
        virtual ~DispersePerception();

        WorldPlayer ourTeamDistIndex(int _iIndex);

        double getMyX();
        void checkVec(Vector2d& _vDisVec);
 };

#endif
```

```
#include "dispersеperception.h"

DispersePerception::DispersePerception(const WorldModel &_wmWorld) : Perception(_wmWorld)
{
}

DispersePerception::~DispersePerception()
{
}

WorldPlayer DispersePerception::ourTeamDistIndex(int _iIndex)
{
	//ClientWorldPlayer cwpClientPlayer = wmWorld.getClientPlayer();
	int iClientNum = wmWorld.getClientPlayer().getNumber();

	//finds out the number of teammates client knows well enough for disperse
	int iArray = 0;
	for (int h = 1; h <= MAX_PLAYER; ++h)
		{
		if (h != iClientNum)
			{
			if ( (wmWorld.getOurPlayer(h).isKnown() == true) || (wmWorld.getOurPlayer(h).getAge() < 6)
)
				++iArray;
			}
		}

	// create array for distances to ball
	WorldPlayer wpTeammateArray[iArray];
 WorldPlayer temp;

 //sets up teammate array
 int iArrayCount = 0;
	for (int k = 1; k <= MAX_PLAYER; ++k)
		{
		if (k != iClientNum)
			if ( (wmWorld.getOurPlayer(k).isKnown() == true) || (wmWorld.getOurPlayer(k).getAge() < 6)
)
			{
				wpTeammateArray[iArrayCount] = wmWorld.getOurPlayer(k);
				++iArrayCount;
			}
		}

	// get distance from the ball and resort the distance array
	for (int i = 0; i < iArray; ++i)
	{
		for (int j = 0; j < i; ++j)
		{
			if ( norm( relPos(wpTeammateArray[j]) ) > norm( relPos(wpTeammateArray[i]) ) )
			{
				temp = wpTeammateArray [i];
				wpTeammateArray [i] = wpTeammateArray [j];
```

```
                                        wpTeammateArray [j] = temp;
                                }
                        }
 }

        if ( iArray >= _iIndex )
                return ( wpTeammateArray[_iIndex - 1] );

        return WorldPlayer();

}

double DispersePerception::getMyX()
{
        return ( wmWorld.getClientPlayer().getAbsPosition().getX() );
}

void DispersePerception::checkVec(Vector2d& _vDisVec)
{
        if ( (wmWorld.getClientPlayer().getAbsPosition().getX() < -0.5 * FIELD_LENGTH + 5) && (_vDisVec.getX()
< 0.0) )
                _vDisVec.setX(0.0);

}
```