A CONTEXTUAL APPROACH TO LEARNING COLLABORATIVE BEHAVIOR VIA OBSERVATION

by

CYNTHIA L. JOHNSON
B.S.E.E. University of Miami, 1986
M.S.E. University of Central Florida, 1990

A dissertation submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
in the Department of Electrical Engineering and Computer Science
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Spring Term
2011

Major Professor: Avelino J. Gonzalez

ABSTRACT

This dissertation describes a novel technique to creating a simulated team of agents through observation. Simulated human teamwork can be used for a number of purposes, such as expert examples, automated teammates for training purposes and realistic opponents in games and training simulation. Current teamwork simulations require the team member behaviors be programmed into the simulation, often requiring a great deal of time and effort. None are able to observe a team at work and replicate the teamwork behaviors. Machine learning techniques for learning by observation and learning by demonstration have proven successful at observing behavior of humans or other software agents and creating a behavior function for a single agent. The research described here combines current research in teamwork simulations and learning by observation to effectively train a multi-agent system in effective team behavior. The dissertation describes the background and work by others as well as a detailed description of the learning method. A prototype built to evaluate the developed approach as well as the extensive experimentation conducted is also described.

DEDICATION

To Uncle Pete, Mac and Mom. Wish you could have been here.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1 INTRODUCTION

Human beings are naturally social. Working in teams to accomplish a common goal is a predominant trait of humans. This is not only evident in team games, assembly lines, and warfare, but also in the simple activities of daily living such as meal preparation. When preparing a meal, particularly a large meal for many people, it is common to have multiple people preparing various parts of the meal. One person might be preparing vegetables while another prepares the meat. The shared goal is to produce a delicious meal by a predetermined time. The outcome of this scenario is dependent upon how well the team of cooks coordinates their efforts.

Being so prevalent in human activities, teamwork has been heavily studied and modeled. Applications of simulated teamwork are indeed numerous. Automating the opposing team in sports games or providing an intelligent group of foes in a video game can add realism and excitement to a game. In military modeling and simulation, a simulated team that behaves like real humans adds realistic foes and teammates, thereby enhancing the effectiveness of training simulations. Additionally, the simulated teams can provide examples of desired behavior, or a standard against which to compare trainee behavior. Ongoing research seeks the best techniques for programming these simulated teams to perform various tasks. One difficulty in modeling teamwork among agents is that the acquisition and representation of the knowledge and expertise of the teams and its members can be difficult, time consuming and expensive.

This dissertation studies methods where the simulated agents can learn the necessary skills and behaviors by observing an expert human team at work. Learning by observation is a technique commonly used by humans. Developing a technique capable of transferring this type of learning to a computer model is desirable.

Learning by observation or imitation has long been studied in humans and animals. [Galef and Giraldeau, 2001] It is debated whether or not the ability to imitate others is a sign of intelligence in animals [Byrne and Russon, 1998], but there is little doubt that it could prove to be a time saving technique for training simulated teams of agents. It is particularly useful when opportunities to derive knowledge from the actual human experts are not available, such as when the expert or experts are absent or otherwise uncooperative (e.g., an opponent or enemy). Learning by observation also offers the opportunity to acquire implicit knowledge that is often not easily captured through other conventional knowledge acquisition techniques. Explicit knowledge includes facts, formulas and rules and is comparatively easy to obtain from an expert. Implicit knowledge, on the other hand, is more esoteric and difficult to articulate and represent. Also known as *tacit knowledge*, it encompasses habits that the expert may not even recognize. This type of knowledge is usually acquired through practice and experience and is often difficult to articulate. One definition of implicit knowledge defines it as knowledge that increases task performance without an accompanying increase in verbal knowledge about the task. [Underwood and Bright, 1996] Another definition is knowledge acquired without conscious knowledge of when and where it was acquired. [Underwood and Bright, 1996] It is sometimes referred to as unconscious memory. [Underwood and Bright, 1996] Learning by observation is one possible way to obtain this knowledge.

In addition to learning tacit knowledge, there are many advantages to learning by observation. These include [Fernlund, 2004]:

- Amount of expert's time needed is minimized. Expert need only demonstrate task, not talk about it.
- Software coding time is minimized as behavior function encoding is automated.

- Development of agent is quicker.
- It is possible to incorporate demonstrations from multiple experts.

Several techniques exist that use learning by observation to train a single simulated entity. Applications exist that have trained agents to drive a car simulator [Sidani, 1994; Fernlund, 2004], teach planning application operators [Wang, 1995], learn to fly [Sammutt, et al, 1992] [Isaac and Sammutt, 2003] drive a simulated tank in formation [Fernlund, et al, 2007] and many others. An extensive review of the recent literature in learning by observation is presented chapter 2. However, learning applications relevant to single agents are often difficult to implement in a multi-agent scenario. [Sycara, 1993] It is possible to individually train individual agents that are part of the team independently from other teammates. This can be a useful first step in a multi agent system, but agents trained in this manner often do not work well together. [Stone, 2007] The effort is analogous to putting a group of human strangers in a team and expecting them to immediately behave and communicate effectively.

1.1 Teamwork in Humans

The question arises as to whether teamwork skills fall into the realm of explicit knowledge, implicit knowledge or both. Before making this determination, it is important to grasp what separates a team versus a group of individuals working in the same place. A good place to start is philosopher Bratman's view of shared cooperative activity [1992] and shared intentions [1993]. It seems intuitive that a group of individuals working as a team would at least have shared intentions. Bratman asserts that shared intentions do not exist unless some basic criteria are met. First, a group only has intentions to perform task J when all members of the team have the intention to perform task J. Second, it must be common knowledge among all team members that everyone in the group is performing task J. Thirdly, the team members must have a joint commitment to mutual support.   Only if these criteria are met can the group be said to have the shared intention to perform task J. [Bratman, 1993]

Using our meal preparation example, two people might be working together in the kitchen, but one might be planning to serve roast for dinner and the other chicken. They are not a team unless they share the intention to serve the same meal. Another example of this might be the difference between two airplanes flying in the same region of space versus a squadron of fighter jets. The two airplanes probably share the intention to avoid a collision, but as they are not necessarily aware of the other airplanes' intention, they are not considered a team. A squadron of fighter jets are considered a team because they share the intention of performing a mission, are aware that the others share that intention and are working together to make the mission occur as planned.

Training an expert team requires understanding and capturing the behaviors or skills that are necessary to perform the tasks assigned to the team. These can be broken into *task skills* versus *team skills*. [Salas, et al, 2000]   Team skills or competencies are individual team member skills that affect team performance such as communications. Task skills are those applicable to the tasks being undertaken by the teams. For example, a team of pilots flying a large aircraft should all have the task skills to read the instruments and use the controls to manipulate the aircraft. However, it is important that they also have the team skills of communication so they are effectively dividing the tasks and handling any emergencies that arise.   Both team and task skills can be classified as generic or specific to the team goals. Generic skills are those that are applicable regardless of the task being undertaken, while specific skills do not transfer well to other tasks. Team skills considered generic include conflict resolution, ability to motivate others, information exchange, planning and flexibility.   Team specific skills include situational awareness, task interaction, and dynamic reallocation of functions and shared problem-model development. The type of task being undertaken by the team will determine the needed skills and competencies of the individual members. However, all team members will need some basic team skills or competencies in order to effectively work together. [Salas, et al, 2000] Defining which skills are most important in impacting the effectiveness of a team is more difficult than identifying common skills. Expertise in task specific skills by individuals comprising the team was found to have a strong impact on team effectiveness, as is having independent self-aware team members. [Mickan and Rodger, 2000] Additional skills and traits found to positively impact team effectiveness were commitment, flexibility, good coordination skills and good communication skills. Team coordination can be defined as interpersonal actions used to complete tasks together. [Mickan and Rodger, 2000] Team communication is defined as exchange of information and interactions that relate attitudes and values. [Loxley, 1997]

The amount and type of the competencies needed for an effective team are determined by the specific type of team that, in turn, is determined by the type of work to be accomplished. [Sundstrom, 1999] Sundstrom [1999] identifies six different kinds of work teams: production, service, management, project, action/performing and parallel teams. The criteria used to differentiate between team are amount of authority, temporary/permanent, external linkage, whole team specialization, and within team specialization. [Sundstrom, 1999]

Management teams refer to teams of managers and their direct reports. The team has responsibility for work under its purview. This type of team has a great deal of authority, is organized per leader and is generally considered permanent. Management teams are moderately specialized. Typically, no two management teams in an organization do the same thing. Project teams are also known as task forces. They are formed to complete a specific task in a definitive time period and typically disband when the task is complete. The members are experts selected for specific skills, and this type of team is highly specialized. An example from industry would be a proposal team. The members are brought together to produce a proposal for a particular job and disband when the proposal is complete. [Sundstrom, 1999]

Service teams perform repeated transactions with customers. These teams typically have very little authority and depending upon the service they provide; have various levels of complexity and authority. A customer service department is a good example of a service team. Parallel teams are highly specialized type of teams that work outside normal channels. They may have characteristics of two or more other types of team. [Sundstrom, 1999]

Production teams repeatedly produce tangible outputs of a particular kind. A common example is a manufacturing assembly line. This type of team has limited authority, is very stable and is moderately specialized. The team is tightly linked with outside teams as well. Action/performing teams are very specialized. Their output is often tangible, but doesn't have to be. The team is very flexible and coordination among the team members is high. The effectiveness of an action/performing team is often limited until team members adjust to one another. The teams are often not stable so adjustment to new team members is dependent upon standardization of skills in individuals and roles. [Sundstrom, 1999]

This dissertation concentrates on learning the skills of an action/performing team and production teams.   Production teams are often simulated to test the effectiveness of a particular configuration of a manufacturing assembly line. It is much less expensive to move equipment and tasks in a virtual assembly line than on an actual factory floor. Gaming has had a huge impact on simulation of action teams. The simulation of an action team makes for a more popular game than the simulation of a service team.   An action team typically performs time-limited engagements against adversaries or challenging environments. Action teams have moderate authority, can be permanent or temporary, and have high external linkage. Furthermore, whole team and within team specialization are high. Effective action teams will have strong task specific skills; however, the team members will also need skills that are considered task generic. For example, the members of the team must have interdependent roles and goals, but they share a fate or outcome. The team members must see themselves as a team and not simply a group of individuals. The team maintains a specialized collective skill specific to the type of engagements they perform. Examples are various military units, stage actors, and public safety incident response teams. They have highly demanding tasks and are capable of quickly adjusting strategies. This type of team needs members with strong competencies specific to the task and team. If the team experiences high turnover, task competencies become particularly important. The amount and type of necessary team skills is heavily influenced by the context and task requirements. . Simulations and team games are often used as training aids to help human teams develop their task and team skills. [Sundstrom, 1999]

Another aspect of modeling teams is modeling the team decision making process. This is particularly important in action teams where it is often necessary to adjust the team strategy quickly. Several psychologists propose that team decision making is best described by a shared mental model of the task and team. [Cannon-Bowers, et al, 2001] [Druskat and Pescosolido, 2002] [Johnson-Laird, 1983] [Mohammed, et al, 2000] The theory of mental models assumes that people organize knowledge into structured meaningful patterns that enable rapid and flexible access to the knowledge and processes necessary to act upon it. This allows for effective interaction with their environment. Effective team members are able to perform this interaction with their environment quickly and efficiently. A shared mental model is a set of organized expectations shared by the team. This does not mean that each member has an identical mental model, but rather that they have shared perception of the team goal and each member's role in the decision making process.

An example of an action team is the crew of a naval vessel determining whether or not to fire at hostile aircraft. This involves multiple team members operating equipment, determining the appropriate response and taking action. Each team member has to understand the operation of the equipment for which they are responsible and how to get information from them. They need to understand what information is significant to the task at hand, what additional information is needed and how to combine that information. Each team member must understand their role in the task and know the capability and role of their team members. In the case of the naval vessel, the radar operators must operate the radar and relay the necessary and relevant information to their officers. The officers must know how to combine that information with their knowledge about rules of engagement and the present tactical situation. The commander must then make the decision to fire upon the hostile aircraft or defer engagement. The weapons operators must then act upon the decision of the officers. All this must be done in a constantly changing environment where the results of the decision could mean life or death.

In this example, a joint goal is the safety and defense of the ship against hostile forces. Each team member must have that goal as part of their shared mental model as well as understand their role and those of the other team members in the decision making process. In addition, in this situation, the team members must have a strong mental model of their individual tasks. For example, the radar operator must have strong understanding of the symbols on the radar screen as well as knowing how to operate its controls. Furthermore, the operator must have adequate skills to effectively filter the information on the screen and communicate only the information his team members need to know. This requires using information from task specific mental model for filtering and knowledge of team members' roles from the shared mental model. [Cannon-Bowers, Salas & Converse, 2001]

These studies on teamwork lead to the conclusion that in order to effectively train teamwork, it is necessary to look beyond merely training the individual team members in their tasks. While it is necessary for the team member to learn their individual task-specific skills, they must also acquire team based skills such as communication, and share common goals. Ideally, each member of the team would have perfect awareness of what the other members know and intend. However, in a simulated system, this is impractical [Barrett, 2007] for all but very small and simplistic teams. It is also not representative of true human behavior. It is possible and desirable for team members to know what the roles and basic skills of their teammates are and use this knowledge to allow the team to work together effectively. An effective team is going to be more productive than a group of individuals working in the same area. It is clear that any type of learning algorithm used to train software agents to work in a team must include learning individual task skills as well as effective team behaviors.

1.2 Teamwork Knowledge and its Acquisition

The skills that contribute to effective teamwork often fall into the realm of implicit knowledge. They are hard to articulate. For example, how do you define "good communication skills"? This is a very subjective area and what could be considered good communication skills in one team might be inadequate in another or even too much communication in a third. Since learning by observation has been shown to allow learning of this type of implicit knowledge as well as the more easily learned explicit knowledge [Sidani, 1994], it seems to be a good fit for training a team. While learning by observation is a natural part of being a human, the translation to a learning algorithm for software agents is not quite as simple. Effective learning requires accurate observation and representation of the expert's behavior before it can be translated into behaviors for the observing agent. Despite the challenges, learning by observation should prove to be an effective means for training a multi-agent system to perform tasks involving teamwork.

The contributions of the dissertation are a new approach to training multi-agent systems to learn teamwork. This is particularly useful to the military simulation and training community for enhancing the existing simulations and training feedback products. This technique would be useful for adding more expertise to automated forces that are used for providing realistic opponents or teammates. In addition, it could be used to train expert agents that could be used to play a simulation exercise along with the trainees and provide valuable feedback on trainee performance as is done in SmartAAR. [Fernlund, et al, 2009]

1.3 Organization of Dissertation

Chapter 2 of this dissertation presents the background of multi agent systems designed for teamwork. Also included are a discussion of various multi agent machine learning techniques and a survey of various methods for learning by observation. Chapter 3 expands upon this background to describe a method of training a multi agent system based upon observed behavior of an expert team. Subsequent chapters describe an agent framework based upon the Collaborative Context-Based Reasoning framework. This framework is built and trained to perform a military team operation. The subsequent chapters discuss the machine learning algorithm developed and evaluate its effectiveness by determining how well the newly trained agents perform compared to the original expert team.

CHAPTER 2 BACKGROUND

This chapter discusses the prior research into simulating teamwork as well as recent research into learning by observation. The background on the theory behind teamwork simulations and existing teamwork agent frameworks are explored. Furthermore, machine learning techniques for multi agent systems are reviewed. Finally, a variety of techniques for learning by observation are examined. This background information is important to understand the research described in this dissertation.

2.1 Human Behavior Representation Architectures

While attempting to simulate human teamwork, it is important to remember that the team consists of humans. Therefore, the agents representing team members must also portray human behavior. The study of human behavior is highly complex. Cognitive architectures or frameworks are the first steps toward creating a complete simulation of human behavior. This task, however, has been termed "possibly the most difficult task humans have yet undertaken". [Pew and Mavor, 1998, p. 8] This section reviews some of the approaches to simulate human behavior using a computer. The approaches range from attempting to recreate the complete human cognitive process to breaking the problem down into contextual pieces.

It is also important to avoid agents that are too perfect. When recreating human behavior, it is desirable to capture their individuality and unpredictability as well. In a simulated team, each member of the team is a simulated agent with unique actions and behavior. The agent is responsible for mapping the current state to the next action to take. This mapping function can take a variety of forms ranging from neural networks to rule-based systems. The following two sections examine the research literature of frameworks designed to simulate human behavior. Each framework has a unique approach to the type and creation of the behavior function.

2.1.1 Cognitively-Inspired Frameworks

A common approach to describing human behavior is to treat it as a black box system. External stimuli are the input. Inside the black box is a variety of functions to process the stimuli, and the resulting output is behavior. This black box can also be referred to as the behavior function of the agent. Adaptive Control of Thought (ACT-R) [Anderson, 1993], SOAR [Jones, et al, 1996], Sandia's Cognitive Framework [Forsythe & Xavier, 2006], COGnition as Network of Tasks (COGNET) [Ryder & Zachary, 1991] are among the many agent frameworks that attempt to create agents with behavior functions based upon human cognition. These frameworks use the study of human cognition as a starting point and develop software agents that simulate the workings of the human brain to varying extents.

ACT-R has a modular architecture, with modules for declarative memory, perceptual systems and motor systems. These modules are all synchronized through a central production system. It is intended to simulate human cognition as closely as possible. The central production system is primarily rule-based with a Bayesian mechanism that learns the utility of a production rule based on its history. New production rules are learned or input while the system is off-line. The system has been used primarily as a test bed to explore cognitive theories. [Best, et al, 2008]

SOAR originally consisted of production rules that acted as long-term memory and a single short-term memory that contained information about the current situation. Learning was done through the addition of new production rules in a process known as chunking, that converts the process of problem solving into rules. While this provided a strong foundation and a useful cognitive agent, the addition of new modules have extended SOAR to expand the types of knowledge represented and perform additional types of automatic learning. [Laird, 2009]

SOAR has also been used as a basis of the Virtual Human project. [Swartout, et al, 2006] This virtual reality simulation is intended for use in researching the issues of achieving human level performance in cognitive systems. It has been successfully used in a system designed to teach leadership skills in high stake social situations such as military interaction with the civilian population. [Swartout, et al, 2006]

COGNET was built as a model of human information processing for specific purposes. The range of purposes has grown over time since it was first created. It does not attempt to capture any particular theory of human thought or behavior. It is intended to provide a framework for cognitive model developers who are not necessarily experts in cognition. Parallel processing of perceptual, motor, and cognitive sub-systems is made possible in part by a shared memory module. There is also the concept of attention that allows COGNET to deal with the competing needs of the parallel processing. Application using agents developed in COGNET include en-route air traffic control, telecommunications operators and naval command and control. [Zachery, et al, 1996]

Sandia's Cognitive Framework (SCF) approaches human behavior representation a little differently. It uses an interconnected semantic network that feeds into context recognition memory. Domain specific concepts are represented as nodes in the semantic network. A perceptual interface provides sensory information into the semantic network simulating the way human senses feed into human cognition. Rather than attempting to model generic human behavior, the approach to SCF has been to model specific individuals. This allows them to validate the results of the modeling against a specific human being's behavior. There is currently no machine learning interface in the SCF. [Best, et al, 2008]

The Cognitive Foundry integrates machine learning into the Cognitive Framework. [Basilico, et al, 2008] A variety of machine learning tools are provided that populate the Cognitive Framework using automatic knowledge capture tools. The decoupling of the learning algorithms from the cognitive frameworks allows for rapid prototyping and quick execution of different algorithms and approaches. The tools include a variety of supervised and unsupervised learning algorithms as well as a statistical validation package. [Basilico, et al, 2008]

coJACK [Evertsz, et al, 2008] is a cognitive agent framework based upon the JACK agent framework. JACK is a based upon the Belief/Desire/Intention agent model and implemented in the Java language. The key programming constructs provided by JACK are events, plans, belief sets and intentions. coJACK augments these constructs with a moderator layer and cognitive architectural constraints and parameters. It is intended for use in resource limited environments. It has been tested in a simple tank game known as dTank. This game was designed to compare various cognitive architectures in a simulated environment. The framework proved successful at playing dTank, but testing is still ongoing on more complicated tasks. [Evertsz, et al, 2008]

The frameworks listed here are just a few of the currently used cognitively inspired frameworks. Other cognitively inspired frameworks used in modeling and simulations include Executive-Process Interactive Control (EPIC) [Meyer & Kieras, 1997], Human Operator Simulator (HOS) [Glenn, et al, 1992], Man Machine Integrated Design and Analysis System (MIDAS) [Laughery and Corker, 1997], Operator Model Architecture (OMAR) [Deutsch, et al, 1993] and Situation Awareness Model for Pilot-in-the-loop Evaluation (SAMPLE) [Baron, et al, 1980].

2.1.2 Context-Inspired Frameworks

Other human behavior representation frameworks are inspired by the idea of context. Humans naturally think in terms of context, one area where humans regularly use context is in conversation. For example, if the phrase, “Where’s a good bank?” was overheard on a city street, the assumption is that the speaker is looking for a bank in which to obtain a loan or deposit his or her money. However, if the same phrase is spoken holding a fishing pole near a river, the context implies that the speaker is looking for a river bank to fish from. The idea behind context-inspired frameworks is that in any given situation there are a limited number of options. By understanding the situation or context, it is possible to encapsulate the necessary knowledge, procedures and actions that are applicable to the situation. If processing natural language and the context is known to be fishing, there is no need to process the word bank in financial terms.

There are many definition of the word “context”. Bazire and Brezillon [2005] built a model of context based upon an analysis of dictionary and web-based definitions of context. The model represents the components of the situation that influence context. They recognize that the context of a particular item is influenced by the user or individual looking at the item, the environment and the observer of the environment. As with many terms, there is no single correct definition of what context is or how it can be determined. The authors did determine that there are six important factors that analyze and define context. These factors are constraint of the context, influence of the context, behavior within the context, the embedded system, nature of the context and structure of the context. [Bazire & Brezillon, 2005]

A contextually based framework has many advantages for agent development. These include:

- The ability to simulate human behavior
- The use of context divides the problem space into smaller, easier to handle problems.
- Several techniques have been developed that automatically generate context-based reasoning agents from observation.

In many ways contexts are analogous to the mental models [Cannon-Bowers, et al, 2001] that humans produce. Experts in any given area are able to process information, determine the context and make a decision based on the perceived context. The knowledge that only certain information is relevant in a particular situation enables the expert to filter the inputs and prune the number of possible actions and facilitate decision making. The ability to determine context is key to obtaining situational awareness. Situational awareness is defined as ".. the perception of the elements in the environment within a volume of time and space, the comprehension of their meaning and the projection of their status in the near future." [Endsley, 1995] Endsley [1995] states that complete human situation awareness have three levels. The first level is the perception of the current state. The state data are then synthesized together to form patterns that lead to an understanding of the situation. In a software agent, this is analogous to current information from the simulation and other agents. [Endsley,1995] These mental patterns are contexts. By knowing the context, comprehension of the perceived state is made possible. This is known as level 2 situational awareness. This is the level most simulated agents are able to achieve. Level 3 situational awareness involves projection into the future and choosing one's action based on all the possible futures. [Endsley, 1995] Because of the infinite number of possibilities, the processing power needed to achieve true level 3 awareness is often prohibitive.

However, context allows for the limiting of possible options and brings simulated agents closer to achieving true human situational awareness.

Several researchers have examined the idea of using context as a basis for models of human behavior. Giunchiglia [1993] implemented a contextual system called Multi-Context (MC) systems. The system formalizes each context in terms of separate language, axioms and rules of inference. Bridge rules allow transition between context as well as allowing concepts to be shared across contexts. The system has been used as a framework for mental representation. [Giunchiglia, 1993]

Turner [1998] hypothesized that context-sensitive behavior is the key to accurately simulating intelligent behavior. He approached adaptive problem with schema-based reasoning. Frame-like schema known as c-schema were developed to provide context sensitive information. The agent determined the context based upon the current situation and selected the appropriate c-schema. The c-schema contains information needed to identify and predict possible scenarios, set behavior parameters, focus attention on appropriate goals, select proper action and respond quickly to events. This approach was tested in applications in medical diagnostics and underwater autonomous vehicle controllers. [Turner, 1998]

Kokinov [1999] defined context as ".. the set of all entities that influence humans (or system's) behavior on a particular occasion". [Kokinov, 1999 p. 205] Kokinov believes that the use of context makes systems more flexible and efficient. This efficiency is gained by reducing the amount of information and number of possible actions to only those important in a particular context. The DUAL architecture was developed using this context. This unique architecture pulls together symbolic and connectionist attributes into a system capable of dynamic reorganization. This architecture was used to design AMBR, a system that simulates the deductive and analogical reasoning aspects of human problem solving. [Kokinov, 1999]

Zibetti, et al [2001] created a model of human behavior called ACACIA (Action by Contextually Automated Categorizing Interactive Agents). They approached the development with a combination of bottom-up (perceptual components) and top-down (cognitive components) processing. Context is determined by already active information from the previous time step along with the current time step's properties. [Zibetti, et al, 2001]

Brézillon [2003] defines three separate aspects of context. These are contextual knowledge, procedural zed context and external context. Brézillon implements context through the use of contextual graphs (CxG). These specialized decision trees are acyclic graphs that have a single source, single sink and series-parallel organization of the nodes. Contextual graphs have been used to successfully implement subway line event management and contextually-aware computing programs. [Brézillon, 2003]

Context-based reasoning (CxBR) is the contextual-based paradigm chosen for use in this dissertation. Because CxBR is a paradigm rather than a framework, it offers flexibility in implementation not available with the other human behavior representation frameworks. CxBR simulates human behavior without simulating all of the human thought processes.   The life of a software agent typically consists of processing information and making a decision on the next action or state. By knowing the current context, an agent can limit expectations as to what is normal in the current context.   When the situation is changed, environmental or internal events can trigger a transition to a new context. [Gonzalez, et al, 2008]

A CxBR agent categorizes rules and function hierarchically. The top level of the hierarchy is known as the Mission Context. Each CxBR agent has a single Mission context. The

Mission can be defined as the process of interacting with the environment and making decisions while trying to accomplish a goal or objective. The Mission Context is responsible for maintaining that objective, defining the criteria for ending the mission, and maintaining the plan. Within a particular Mission, various situations are categorized into Major Contexts that can contain Sub-Contexts. The Mission contains universal rules that are applicable to all contexts and contains a list of all the possible Major Contexts applicable to the mission. These universal rules, sometimes called *universal sentinel rules*, are typically a set of actions triggered by a particular state or data transition. These universal sentinel rules are checked by all the contexts because they define conditions that should result in a particular action regardless of the current context. Each context contains additional rules and functionality specific to that context, also known as *action rules*, and rules governing the transition from one context to another, known as *transition rules*. Sub-contexts represent a lower level of abstraction than the Major Contexts, but operate in essentially the same manner with the exception that context transitions are only between Sub-contexts of the active Major Context or back to the Major Context. Each context is an object-oriented class with its own set of attributes and methods relevant to that context. The flexibility and encapsulation of the contexts make CxBR agents lightweight and easy to implement. Figure 2-1 shows a conceptual model of a CxBR agent.

2.2 Models of Teamwork

As discussed in Chapter 1, a group of individuals or software agents does not create a team. While using software agents based upon human behavior is a good starting place, additional elements are necessary to create an effective team of agents. Theories of agent teamwork have been proposed in the literature. These provide formal requirements for building teams of agents. Two of the most popular are *Joint Intention Theory* [Cohen & Levesque, 1991] and *SharedPlan*

[Grosz & Kraus, 1996]. These two theories are broadly reviewed and compared to the psychological models discussed in the first chapter.

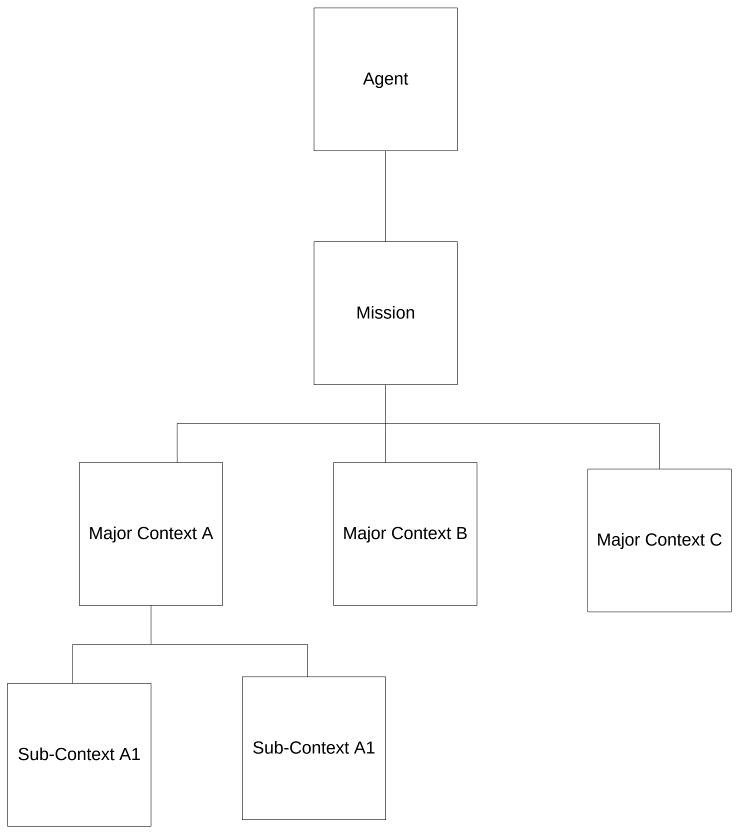


Figure 2-1 Context Hierarchy for a Basic CxBR Agent

Cohen and Levesque [1991] applied the idea of teamwork to software agents that must either work together or with human agents. This theory is known as Joint Intention Theory (JIT). This theory is concerned with the design of software agents that have to interact with other agents, either real or simulated. A joint intention is defined as a joint commitment by a team to completing a particular action. In order for a group of agents to be considered a team, they must have a joint commitment to a goal. This is referred to as a *joint persistent goal*. The team has a joint persistent goal to achieve P, where P is the completion of the team action when the following is true:

- *All team members believe P to be false*
- *All team members have a mutual goal to make P become true*
- *All team members believe that their team members also have a goal to make P true until it is mutually believed to be true, unachievable or irrelevant. [Cohen and Levesque, 1991]*

The psychological theory of shared mental models [Cannon-Bowers, et al, 2001] ties in nicely with Joint Intention Theory. The knowledge of what P is and its current state would be part of the shared mental model of the team. Each team member's role in achieving P would also be part of the shared mental model. The specific skills necessary to perform each and every task would not be part of the shared mental model; rather, each team member would have a unique mental model of their skills and processes necessary to perform their tasks in addition to the shared mental model of the team.

SharedPlan [Grosz & Kraus, 1996] focuses on the collaborative plans of the agent as well as their intentions. Like JIT, it can be used as a specification upon which a team of agents can be built. A group of agents are said to have a Shared Plan to perform A if the following holds true:

- *Each individual intends that the group do A*
- *Each individual has a mutual belief in a (partial or full) plan for A*
- *All individuals have a plan for the sub-acts involved in A*
- *All intend to succeed in doing the sub-acts*
- *All commit to group decision-making processes aimed at completing the plan. [Grosz & Hunsberger, 2006]*

Agents have a shared commitment to a goal as well as a shared plan on how to achieve that goal. The shared plan breaks down into individual plans for each agent. Each agent looks at each goal in the plan for what it is capable of bringing about and what the group is capable of bringing about. The idea is that collaborative activity is brought about by the individual actions and domain actions of each agent. [Grosz and Kraus, 1999]

Like JIT, SharedPlan also fits well into the idea of shared mental models, with the shared plan and knowledge of other agent's capabilities fitting into the shared mental models. In addition, it recognizes the difference between domain skills and team skills discussed in Chapter 1. Obviously, JIT and SharedPlan have some similarities, as both require that agents have an intention to perform their task. SharedPlan focuses more on the planning aspect of the task than JIT. Although developed independently, both theories focus on shared intention as a basis, and contain elements useful for developing an agent framework. More importantly, both have elements that can be linked with the psychological basis of human teamwork. This is important because we are attempting to simulate human teamwork behavior. Any framework used to simulate human teamwork should have elements that echo the actual human behavior.

2.3 <u>Teamworking Agent Frameworks</u>

Several agent frameworks geared toward simulating teamwork have been developed using JIT as a starting point. Two of the earliest frameworks were STEAM [Tambe, 1997] and GRATE [Jennings, 1995]. STEAM, a Shell for TEAMwork, is a multi agent framework that adds a detailed communication model to the SOAR [Laird, et al, 1987] agent framework. To accommodate teamwork, STEAM adds team operators and team representation structures to SOAR. These additions allow for the monitoring of the team performance. STEAM incorporates aspects of both JIT and SharedPlan. It primarily uses JIT for specification, but the researchers found issues revolving around communication among teammates, coherency of the path forward, and ability to replan when joint intention is seen as unachievable. To address these issues, aspects of SharedPlan were used particularly in the area of replanning. The hierarchical structures used to implement STEAM were also very similar to SharedPlan. [Tambe, 1997] This generic framework for teamwork has been applied to a variety of domains. These include combat air missions [Hill, et al, 1997], robot soccer [Kitano, et al, 1997], and rescue response [Scerri, et al, 2003]. More recent updates to STEAM include TEAMCORE [Tambe, et al, 1999] and Machinetta [Schurr, et al, 2006]. These updates take advantage of the advances in computing power and new computer languages that have emerged through the years to create more lightweight and flexible agents.

GRATE (Generic Roles and Agent model Testbed Environment) is another agent framework developed using JIT. [Jennings, 1995] GRATE was built upon the concept of joint responsibility, essentially joint intention with modifications to handle planning failures or unplanned events. GRATE is built to address the following points of teamwork: [Jennings, 1995]

- *A joint goal is shared by all agents*
- *All agents wish to cooperate with each other to attain the joint goal*
- *All actions performed in the context of the joint action are interdependent*
- *Agents must have a means to monitor their commitment.*

In order to address these points and create a general purpose framework for cooperation and situational awareness, GRATE agents consist of two components; there is a cooperation and control layer that is separate from a domain-level system. The cooperation and a control layer are intended to provide the generic cooperation and representation structures needed by all forms of teams while the domain level provides the details of the particular application. Problems are viewed as atomic units of processing known as tasks to the cooperation layer. The framework has been used to implement a variety of manufacturing control systems. [Jennings, 1995]

There have also been several applications based upon the SharedPlan theory. These include Collagen [Rich & Sidner, 1998], a distance learning tool [Ortix & Grosz, 2002], and an e-commerce tool [Hadad & Kraus, 1999]. However, they are based on SharedPlan theory, and not a common framework or architecture. Nevertheless, some work has been done towards developing a SharedPlan agent framework called MIST [Nguyen & Wobcke, 2006]. None of these agent frameworks are particularly well suited for learning teamwork by observation, as the frameworks are set up assuming that the agents will use the predefined teamwork modules for the communication and coordination of the team.

A few agent prototypes have been developed that explicitly implement the concept of shared belief or shared mental models for simulating teamwork. JTEAM is one such prototype. [Hsu, et al, 2003] JTEAM implements the idea of a team goal that translates into a team plan which can be chosen from a list of applicable plans based on the goal. The objective of the team plan is to translate the joint goal into sub-goals that are implementable by one agent. The team leader is responsible for apportioning the sub-goals out to the various agents. There can be only one team leader at a time, but the team leader can change over time. The team has a set of team beliefs and every agent has a copy of these beliefs. Only the team leader can freely change the team beliefs to avoid an inconsistency of team beliefs. [Hsu, et al, 2003]

CAST, Collaborative Agents for Simulating Teamwork, is another prototype that implements the idea of a shared mental model. [Yen, et al, 2006] Designed to study teamwork-related issues, CAST agents have individual behavior components and components for maintaining a team-shared mental model. All the agents are programmed with a commitment to maintaining the shared mental model through outgoing communication of their own status and processing incoming communication about other team members' status. [Yen, et al, 2006] Another example of research into implementing shared beliefs among a large multi-agent team using a token-based algorithm. [Velagapudi, et al, 2007] In this research, the communication of data pertinent to the shared situation awareness of the team is encapsulated as a token and forwarded to a teammate. That teammate integrates the information with its own set of beliefs and determines whether the information is still relevant. If so, the token is passed on to another teammate. This process is repeated until an agent determines that the information is no longer worth propagating. Using lessons learned from token ring networks, a token-based belief sharing policy was developed. [Velagapudi, et al, 2007] Unfortunately, the research is still in its early stages and no prototype is available to try this method of information sharing.

Other multi-agent frameworks have found widespread acceptance are JACK agents [Howden, et al, 2001] and JADE [Bellifemine, et al, 2007] agents. Both are based upon the Java language. By using Java, the agents are not limiting themselves to a particular operating system or platform in the way that some languages such as C# do. A few older frameworks like AgentBuilder [AgentBuilders, 2006], FIPA-OS [Poslad, et al, 2000] and Zeus [Nwana, et al, 1999] have found widespread acceptance and are readily available for download, but none of these have been updated for use on contemporary operating systems. FIPA-OS is compliant with agent specifications developed by FIPA, the Foundation for Intelligent Physical Agents. Another less readily available FIPA-compliant framework is SAGE. [Ahmad, et al, 2005] SAGE is not readily available, but researchers in Asia have used SAGE to develop a multi-agent teamwork system based on the teamwork exhibited by honey bees. [Sadik, et al, 2006]

2.4 Collaborative Context Based Reasoning

In this dissertation, the collaborative agent framework chosen for implementation is Collaborative Context-based Reasoning (CCxBR). CCxBR was developed to implement teamwork. Barrett [2007] formalized CCxBR in terms of joint intention and related it to the popular Belief-Desire-Intention (BDI) model. [Georgeff, et al, 1999] In CCxBR, an agent is always aware of its current Mission and the current context, whether it is a Major Context or a Sub-context. [Barrett, 2007]

CCxBR builds upon the work of Johansson, who implemented collaborative behavior between CxBR agents using a shared Mission context. [Johansson, 1999] This Mission context is a form of joint intention as the Mission context contains the high level goal of the agent. However, this did not provide any sense of shared situational awareness or means of coordination between the agents. To address this issue, a teamworking class was introduced to the framework that focused on communication between collaborating agents. [Johansson, 1999]

Johansson's incorporation of the teamworking class included the creation of a team mission known as the team mission context shared by the group of collaborating agents. This team mission included the specification of sub-goals for the team members. This addition made more of the information needed for effective teamwork available to each of the team members. Among the information included was team member status, team mission, status of the mission objective, and the role of individual team members. It also provided a means of communicating status changes and mission objective changes that could change individual sub-goals. [Johansson, 1999]

CCxBR implements Johansson's teamworking class using a Team Context. Figure 2-2 shows the basic CxBR with the addition of the Team Context. This Team Context is shared among the CxBR agents functioning as a team. It contains information about each agent's context as well as the joint goal and status of the joint goal. The basic concept behind CCxBR is that team members can easily maintain coordination by communicating their current context to each other via the Team Context. By virtue of knowing the context of another agent, an agent can reasonably predict its actions.

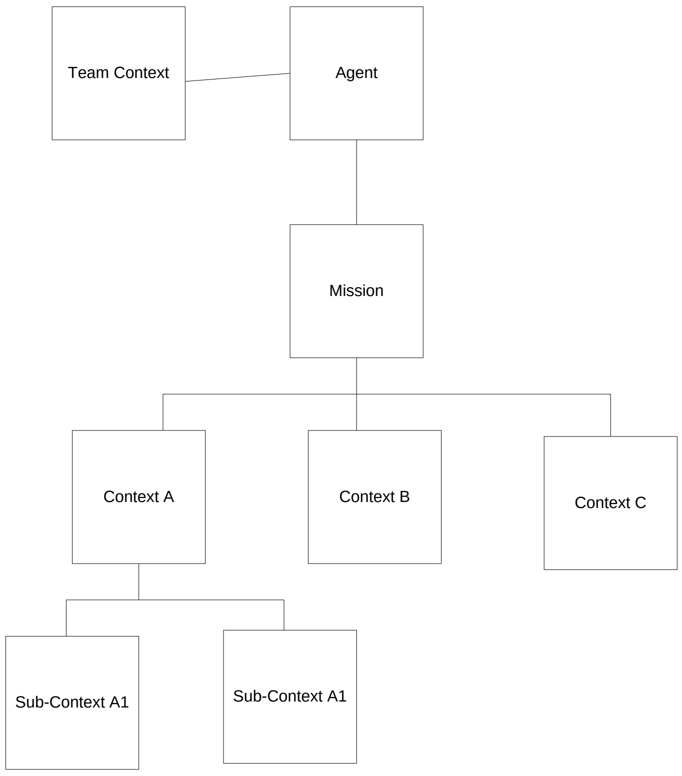


Figure 2-2 CxBR agent with Team Construct

Barrett built three separate CxBR prototypes designed to perform the same basic plays in a simulated soccer game. The first prototype consisted of a team of basic CxBR agents with no accommodation for teamwork. This version simply added a shared team context containing the current context of the mission. The second prototype called "CxBRwithJIT" used the same basic agent, but added reasoning to allow each agent to infer their teammate's context based upon position and state of the game. The third prototype implemented CCxBR and implemented explicit communication of each agent's context among the teammates. Experimentation showed that while the CxBRwithJIT prototype performed better than CxBR with shared Mission, the CCxBR prototype was the most effective at teamwork. [Barrett, 2007]   This dissertation uses the third prototype as the paradigm of choice for a team displaying collaborative behavior in this dissertation. The shared Mission and team context will provide the necessary shared mental model and link to JIT needed to effectively duplicate human teamwork behavior. The individual agents will have the ability to learn and duplicate individual task skills by developing their own Major contexts. Conceptually, Figure 2-3 shows the team members sharing the Team Context.

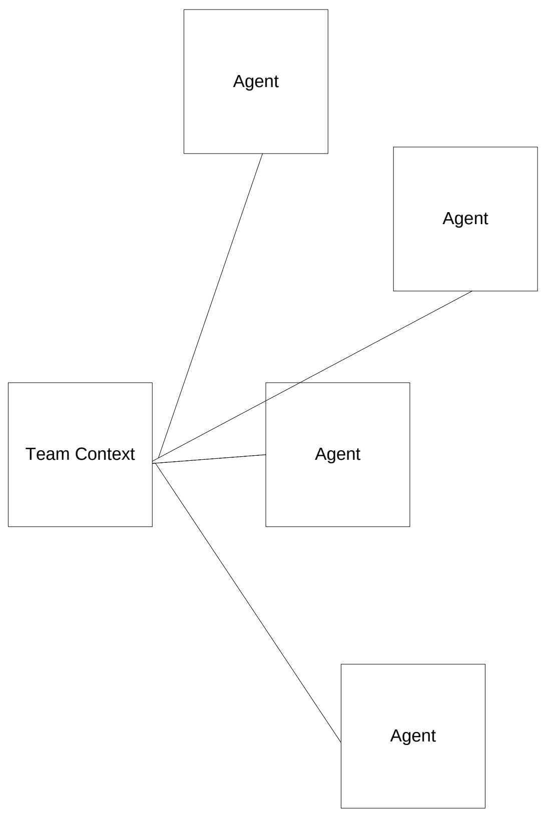


Figure 2-3 CCxBR Agents Sharing Team Context

2.5 Machine Learning Techniques for software agents

A variety of machine learning techniques exist to train software agents. These techniques can be divided into two categories: supervised and unsupervised learning. Supervised learning consists of presenting the learning algorithm with training data reflecting the inputs and expected outputs. The learning technique must analyze or classify the data, and create a behavior function for the agent from the data. Examples of supervised learning are Gaussian mixture models, artificial neural networks, and Bayesian statistics and support vector machines. Unsupervised learning, on the other hand, provides only input data, and the learning algorithm must explore the problem space, developing a behavior function based upon the results of that exploration. Reinforcement learning techniques typically fall into the unsupervised learning category although they are sometimes given their own category. [Russell & Norvig, 2003] Learning by observation falls into the supervised learning category although the inputs and outputs are not explicitly defined. They must be extracted from a sequence of data depicting correct behavior over time. The example behavior and its results are input into the learning algorithm and some sort of behavior function is output. The format of the behavior function will depend upon the type of agent being developed. A rule-based agent will develop a set of rules governing its behavior. An agent governed by a Markov Decision Process will develop a policy that maps state to action and state transitions. A Neural network based agent will develop connections and connection weights to each neuron.

2.5.1 Teamwork Recognition

A topic related to learning teamwork is that of teamwork recognition. Recognizing teamwork from temporal-spatial data is quite useful as a preprocessing step in learning-by-observation. It provides a means of translating video and simulation data into a data structure easily processed by the learning- by- observation algorithms. Teamwork recognition is a form of behavior recognition. One approach to the behavior recognition is template-based reasoning. [Drewes, 1997] Similar to case-based reasoning, template-based reasoning uses templates consisting of a series of attributes. The attributes in each template are given weighted values indicating the importance of the presence of that attribute. An entity's attributes are compared to the stored templates using a simple weighted sum. If the sum is large enough, the entity is performing the behavior represented by the template. Template-based reasoning was found to be useful for automated performance monitoring. [Drewes, 1997] Gerber [2001] enhanced this approach by using neural network-based template attributes. The neural networks were built using learning-by-observation techniques. The resulting system was used to decrease bandwidth use when synchronizing behavioral models of a human–controlled vehicle with the actual human controlled vehicle's actions. [Gerber, 2001]

Sukthankar and Sycara [2006] used team spatial templates and spatially invariant Hidden Markov Models to recognize specific maneuvers by a simulated two man military team. Predefined templates of spatial positioning were created and compared against the positions of a team controlled by human players of the simulation. A highly efficient randomized search algorithm known as RANSAC (Random Sampling and Consensus) was used to do the comparison. In cases where the team behavior could not be determined strictly from spatial coordinates, it was necessary to use temporal data to improve the detection algorithm. The data was transformed into a canonical reference frame defined by the team's motion and a set of Hidden Markov Model classifiers were used to further classify the behavior and help differentiate between three spatially similar behaviors. [Sukthankar & Sycara, 2006]

Luotsinen, et al [2007] expanded upon this work by adding the capability to add new Hidden Markov Models (HMM) based upon representative examples. A data editor was created to isolate the representative examples from a large dataset acquired from a real world warfare exercise involving eight tanks organized into two opposing platoons. Two different techniques for detecting Hidden Markov Models were used to create the new classifiers. The first technique was the Baum-Welch algorithm [Baum, et al, 1970] for statistically determining HMM's and the second was the Segmented K-Means algorithm [Juang & Rabiner, 1990]. The HMM classifiers obtained were approximately 70 to 80% successful in classifying the representative behaviors obtained. [Luotsinen, 2007]

Similar work has been done to recognize patterns in sports teams. Han and Veloso [2000] used HMMs to recognize team behavior in simulated RoboCup teams. Intille and Bobick [1999] were able to recognize the execution of a particular play from video of a football team. For each play they wanted to recognize, a temporal structure description of the behavior was created. A belief network similar to Bayesian belief networks was constructed for each player from this description. Each network was then compared to a video frame using an agent creating a multi-agent system attempting to recognize the behavior of the team. By evaluating the system at each video frame, they were able to recognize the desired play about 80% of the time from a video containing 29 different plays. [Intille & Bobick, 1999]

2.5.2 Pervasive and Context Aware Computing

Behavior recognition also plays an important part in context aware and pervasive computing. This area investigates multi-agent software systems meant to encourage teamwork between software agents and humans as part of pervasive computing. [Schurr, et al, 2005] [Chalupsky, et al, 2002] Pervasive computing applications typically take the form of an assistant to a human and often tie into the human's mobile computing device such as a smart phone. [Lee, et al, 2007] The applications are called context aware because the behavior function is often dependent upon the context of the user. While these applications typically include the ability to learn a behavior function based on the human user's action, the actions the agent can take are usually small in number and decision making fairly simplistic. For example, a scheduling agent may choose to postpone a meeting by 15 minutes when its user has not arrived at the meeting by five minutes after the start time. [Chalupsky, et al, 2002] While pervasive computing reflects a different type of team than the action team this dissertation is simulating, some of the machine learning techniques used in pervasive computing are quite similar to those explored in the following two sections.

2.5.3 Multi-Agent Machine Learning

Multi-agent systems such as those used to simulate teamwork insert additional complications to the learning process. The predominant approach in the literature seems to be the use of reinforcement learning in training multi-agent systems. Reinforcement learning is a learning situation where an agent attempts to improve its behavior by rewarding or punishing actions taken. Reinforcement learning blends supervised and unsupervised learning. The learning can be considered unsupervised because the learning is done while the system is on-line and running its normal function. It could be said to be supervised because the desired end goal is known a priori and the reward functions are based upon that fact. Various algorithms are used to determine the reward or punishment given during this trial and error approach to the problem.

Theoretically, single agent reinforcement learning allows for an eventual convergence to a single optimal solution in a stationary environment. [Ng, et al, 1999] However, it is nearly impossible to obtain a stationary environment in a multi-agent learning environment. Each of the agents is constantly evolving. The prediction of another agent's behavior is difficult and direct communication of current status with each agent can be difficult and expensive. [Zhang, et al, 2009] Reinforcement learning typically maps the state space vector to an action vector. In multi-agent systems, both of these vectors can become quite large and unwieldy. This typically increases both processing and learning time. This difficulty has resulted in several modified approaches to the problem of reinforcement learning in multi-agent systems.

Zhang, et al [2009] use the idea of a supervision framework known as Multi-Agent automated Supervisory Policy Adaptation (MASPA) to accelerate the learning process. MASPA uses heuristics to guide the learning process. The heuristics fuse the activity of lower-level agents and generate supervisory information to guide and coordinate the learning process. The supervision algorithm was proven to significantly improve learning time over conventional reinforcement learning in two separate application domains. [Zhang, et al, 2009]

Case-based reasoning has also been used to improve the performance of reinforcement learning in a multi-agent system. [Jiang & Sheng, 2009] Case-based reasoning is used to aid mapping of the current situation to a proposed action. The reinforcement learning algorithm is used to update how well the chosen case matched the situation. Jiang & Sheng [2009] successfully experimented with the technique in a dynamic inventory control application. Molineaux, et al [2009] use plan recognition to improve the performance of a case-based reasoning based reinforcement learner. By using a clustering algorithm to learn opponent's action, Molineaux, et al were able replace low-level features with a prediction of the opponents plan. This sped up the learning process of their agents that were simulating an American football team, significantly from the original application that used only case-based reasoning reinforcement learning. [Molineaux, et al, 2009]

The annual RoboCup competition has inspired many multi-agent learning algorithms. [White & Brogan, 2006] [Stone & Sutton, 2001] [Kalyankrishnan et al, 2009] One group has experimented with batch reinforcement learning algorithms to improve their team's ability to learn. [Riedmiller et al, 2009] Multi-layer perceptors (MLPs) and regression algorithms are used to predict the world model rather than using the sensed environment. Action decisions are based upon predictions of the future rather than the current sensed environment. The algorithm was used in three areas within the context of the Brainstormer's RoboCup competitive team. This team has been competitive in various leagues of the RoboCup competition since the inception of RoboCup in 1997. [Riedmiller et al, 2009]

The biggest problem with reinforcement learning is that the learning is done while the agents are running in the target system. During the learning process, the agents' performance typically starts out as very poor and improves as the agent experiences more of the aspects of the environment. While this poor performance is usually acceptable in a simulated domain, it is not acceptable in any domain where poor performance could damage equipment or endanger lives. Additionally, reinforcement learning can produce behavior that is too perfect rather than human-like. [Stein, 2009] This is useful when trying to improve upon human behavior, but less useful when the goal is reproducing human behavior.

2.5.4 Learning from Observation

An alternative to reinforcement learning is learning from observation, also referred to as learning by observation. The process of creating the behavior function of a simulated agent to perform a specific task involves knowledge engineering. In knowledge engineering, information on the task is acquired from expert or experts in the task domain. This information is transformed into usable form such as rules for rule-based systems or code for hand coded functions. The knowledge engineer is the person responsible for obtaining information from the expert and transforming it into the desired form. Computer-aided tools such as CITKA (Context-based Intelligent Tactical Knowledge Acquisition) [Gonzalez, et al, 2002] and PEGASUS [Shaw & Gaines, 1983] have provided aids to automating the process of interviewing the expert. However, even when computer-aided tools are used; it is a long and expensive process that generally fails to acquire implicit knowledge. Observational learning techniques can overcome these difficulties.

A number of techniques have been used to develop behavior functions for software agents using learning by observation. Another term used in the literature is behavior cloning. [Isaac & Sammut, 2003] In addition, robotics researchers have developed related techniques for learning from demonstration. These terms are often used interchangeably. The primary difference between them is that during learning from observation, no interruption of the observed party occurs. The learning algorithm must create an implicit association of input data with expected results without input from the observed party. In learning by demonstration, the observed behavior is explicitly designed provide input into the learning algorithm. The techniques used are typically applicable for both agents and robots so in this section, we will discuss techniques and algorithms developed for both learning by observation and learning from demonstration.

The goal in learning from observation is to emulate the behavior chosen and create an agent or robot behavior function simply by observing original human behavior. The idea is for the target agent to behave nearly identically, given the same or similar circumstances. The simple act of observing can be quite difficult, particularly if the behavior chosen is human behavior. Humans simply do not input and process data in the same way as a computer or robot.

Therefore, the first step in learning from observation or demonstration is to observe the behavior and classify it into data that are easily processed by a learning algorithm.

Depending upon the type of behavior being learned, this can be a complicated process, particularly when attempting to emulate human behavior. Sammut modified flight simulator code to output the action and situation to a log file to obtain data with which to teach an agent to fly the simulator. [Sammut, et al, 1992] Henninger [2001] observed simulated tanks performing a tactical maneuver known as a Road March. This data was used to train a back-propagation feed-forward neural network that would predict control commands to the tank in order to reduce band-width across the simulation network. Sidani [1994] used a series of neural networks to create a knowledge base capable of driving a simulated car from observed data.

Once the observed behavior is placed into a computer readable form, it must be classified into behavior functions. The behavior function generated from the classification of the data can take a variety of forms. A popular behavior function in robotics is typically a mapping of the current state information to a particular action or actions known as a policy. The action selected is likely to be what Bentivegna and Atkeson [2001] call primitives. Primitives are defined for each application, in this case playing air hockey. For air hockey, typical primitives included left hit, right hit, straight hit, and prepare to hit. The primitives were actually a series of commands to the hardware of the robotic arm. The commands to perform a particular primitive did not change, so these were not learned from observation. The behavior function was trained to choose a particular primitive for the robotic arm from the human player demonstration. [Bentivegna & Atkeson, 2001]

Once the behavior function is learned, it is often necessary to generalize or add to the knowledge of the agent. It is possible when performing a learned behavior in a new environment that a situation or world state occurs for which there no action policy or rule is associated. Bentivegna & Atkeson [2001] used a k-nearest neighbor algorithm for this situation for their air hockey playing robot. The k-nearest neighbor algorithm computes the closest learned state to the current state and chooses the action learned in that state. This works well when there is a relatively small state vector. However, the larger and more complex the state vector; the more difficult it becomes to determine the nearest neighbor. One possible alternative technique is to request additional demonstrations and continue learning. [Chernova & Veloso, 2007][Stein, 2009]

A similar approach was used to teach a robot to navigate a maze. [Chernova & Veloso, 2007] This approach was slightly different in that the behavior function was learned by allowing the expert to directly control the robot through the task. The observation of state was made along with the expert's action and a behavior policy was developed from the combination of state and action. Because human input can often be contradictory for the same input, Gaussian mixture models were used to classify the inputs of the human experts before adding them to the robot's policy. A model was developed to represent each possible action that the robot could take. The human inputs were classified into one these models. In order to minimize the amount of demonstration needed, the robot could request human demonstration when attempting to utilize the learned policy in a new situation. [Chernova & Veloso, 2007]

In software agents that learn from observation, developing rules for an expert system and training neural networks are popular techniques. Sammut chose induction to develop decision trees as the basis of his behavior function while training an agent to control an airplane in a flight simulation. A separate tree was trained for each of the various controls of the aircraft such as altitude, pitch and heading. [Sammut, et al, 1992] Later the researchers discovered that these trees did not generalize to similar situations well. To allow for more generalization, Isaac & Sammutt [2003] added a section to learn goals of the pilot as well as control the various aspects of the flight. Neural networks have also been trained by observation to serve as part of behavior functions. [Stensrud & Gonzalez, 2008] [Stein, 2009]

Learning by observation or demonstration has also been used to build case libraries for use in a case-based reasoning system. [Ontañón, et al, 2009] Ontañón, et al created a case-based approach to planning for real-time strategy games. The real-time case-based planning system called Darmok 2 uses snippets and episodes observed from a human player to evolve a case library. The demonstrations are used to derive plans represented as petri-nets. At run-time, the petri-nets are evaluated for their similarity to the current situation. The approach was evaluated with three strategy games with different skills needed. The results varied from human level performance down to very poor performance in the various games. [Ontañón, et al, 2009]

A similar approach was used to create simulated soccer players capable of imitating existing simulated players. [Floyd, et al, 2008] The imitated players were from previously winning RoboCup simulated league teams. The observed players were rule-based, stateless agents. The log files from games played by the observed players were parsed and used to develop a library of cases. A k-nearest-neighbor algorithm was used to determine the best-case matches during the game. A genetic algorithm was used to find the optimal weights for the k-nearest-neighbor algorithm. Because of time constraints, it was necessary to restrict the number of cases in the case base to a number of cases determined by dividing the available time by the time to perform a single k-nearest-neighbor distance calculation.   This method was successful in training agents in individual behaviors, but failed to adequately capture collaborative behavior among agents. [Floyd, et al, 2008]

2.5.5 Learning by Observation through Context-Based Reasoning

Several techniques have been used to develop a CxBR (Context-based Reasoning) agent from observed behavior. In order for a CxBR agent to function properly, it must know how to behave in a certain context (action functions and rules) and when it is appropriate to switch to a new context (transition rules). Any learning strategy used with CxBR must evolve both types of knowledge in order to be effective. Because CxBR is a paradigm rather than specific implementation, the action rules and transition rules need not be literal rules in an expert system. The very nature of CxBR can facilitate learning behaviors from observation.

Fernlund [2004] created a CxBR agent capable of driving a simulated car through an urban setting using learning by observation. He used genetic programming as his learning strategy. Fernlund modeled human driving behaviors after observing them in a simulated car. The driving simulation was instrumented to allow the human drivers' action to be logged for use in training. The contexts for the driving tasks were predetermined and the data partitioned to match the predetermined context. A Genetic programming algorithm was created called Genetic Context Learning or GenCL. GenCL was used to evolve the rules controlling the activation of the contexts and the actions of the driver within each context. A population of instruction trees that could be translated into C code was developed. Crossover and mutation operators were created to combine the instruction trees in various configurations. The instruction trees were converted into compilable source code. A simulator known as the MicroSimulator was used a fitness function. It contained minimal operational requirements to evaluate the performance of an evolving agent. Each individual in the population was run through the MicroSimulator and the fitness score was generated based on a comparison with the original observed data in the same situation. The resulting agents performed the initial task well and were able to generalize the skills learned in similar scenarios. [Fernlund, 2004]

Fernlund was able to show the flexibility of GenCL by creating CxBR agents from observed live military training data. [Fernlund, et al, 2009] The data were collected from two opposing tank platoons using a Deployable Instrumentation Training System (DITS). In DITS, each tank was instrumented with GPS to accurately record its location and firing events were simulated with laser attachments to the weapons. The information collected by DITS instrumentation was transmitted to a central server where it was logged for future analysis. Using the DITS data and GenCL, Fernlund, et al [2009] were able to create CxBR agents able to simulate the tank platoons' movements. These simulations were used to aid in the assessment or after-action review of future live training exercises. [Fernlund, et al, 2009]

Trinh [2009] expanded upon Fernlund's work by learning the context structure by observation in addition to the action function and transition rules. Trinh actually created two version of an observer module that automatically parsed the observed data into contexts. The observed data were the same logs from the human drivers observed in Fernlund's work. [Fernlund, 2004] Using the same simulated car and virtual world as Fernlund, he merged his work with GenCL to create an agent capable of navigating the simulated car through the virtual world. The first approach named Contextualized Fuzzy ART (CFA) used the data point clustering technique known as Fuzzy ART. The clustering technique grouped the observational data into contexts using instances of time as the basis for clustering. This provided a benchmark against which the second approach could be evaluated. [Trinh, 2009]

The second approach known as the Context Partitioning And Clustering (COPAC) method consisted of a combination of clustering and partitioning algorithms. The first partitioning algorithms were known as standard sequence partitioning and fuzzy partitioning. Two clustering algorithms, k-means and similarity clustering were used in conjunction with the partitioning to create four unique algorithm combinations. The resulting contexts are then fed into GenCL to complete the development of the final agent. The generated contexts were quite different from the original human generated contexts, but were still meaningful and usable by the GenCL algorithm. Although, the process of creating the agents was computationally expensive and time-consuming, the algorithms produced agents that behaved nearly as well as the original driving agents developed by Fernlund. [Trinh, 2009]

Stensrud & Gonzalez [2008] used learning by observation to learn the criteria for context transitions in a CxBR. He observed a player in a computerized game of strategy. Sequences of observations were associated with the human action taken and grouped into training data. These observations were mapped by the FAM/Template-based Interpretation Learning Engine (FAMTILE). FAMTILE combines the Fuzzy ARTMAP (FAM) neural network clustering technique with template-based interpretation. [Stensrud & Gonzalez, 2008]

FAMTILE is able to infer the context of the observed human actor and then map that context to the environment. The inference of context is done by the template-based interpretation and the mapping utilizes the FAM clustering network. The input and output patterns governing context switches is generated and used to train a neural network to recognize observation patterns and map them to contexts. The algorithm was tested on maze navigation games and Texas Hold'Em Poker computerized games. The authors concluded that FAMTILE is an adequate technique for learning high-level behavior. It also proved to have an excellent track record for predicting subjects' actions. This could be extremely useful in gaining a perspective of why the human actor is doing what he/she is doing. [Stensrud & Gonzalez, 2008]

Stein [2009] created a multi-modal learning system that incorporates learning by observation with instructional and experiential learning. He created software agents to replace human players in computer based games. The goal was to create optimally performing agents that incorporated human like behavior. A hybrid learning algorithm was chosen to implement the various modes of learning. For learning by observation, the chosen algorithm was neuroeveolution. Neuroevolution uses genetic algorithms to evolve neural network weights and topologies. The resulting neural network functions as the behavior function of the agent. The initial neural network was created during the learning by observation phase using an algorithm based upon Neural Evolution of Augmenting Topologies (NEAT). [Stanley & Miikkulainen, 2002] This was combined with an optimization algorithm known as Particle Swarm Organization (PSO) [Kennedy & Eberhart, 1995] to create a learning system with three stages of learning; observational, instructional, and experiential. This unique hybrid algorithm is known as Particle swarm Intelligence and Genetic programming for the Evolution and Optimization of Neural networks or PIGEON. During experimentation, agents developed using learning from observation were only compared against agents using various combinations of the three techniques. While the observation only agents performed well in most domains, the agents developed using all three techniques sequentially scored consistently higher in the test domains. The additional learning provided by the experiential and instructional modes was especially useful in creating agents able to generalize the capabilities to new scenarios. [Stein, 2009]

2.5.6 Learning from Demonstration Approaches

Learning from demonstration attempts to map world states with actions based upon examples or demonstrations provided by a teacher. This mapping is also known as a policy. This enables the robot or software agent to select an action based upon the current world state. Examples are defined as sequences of state-action pairs recorded during the teacher's demonstration of the desired behavior. Other techniques typically learn a policy from experience as in reinforcement learning. In the robotic community, three core approaches have been developed for policy derivation in learning from demonstration. These are mapping function, system model and plans. In a mapping function approach, the function mapping from state to action is approximated directly from the demonstration data. In a system model approach, a model of the world (T(s`|s, a) is determined from the demonstration data. Often a reward function (R(s)) is also derived to improve the model during execution. The third approach uses the demonstration data to learn rules that associate each action with a set of pre and post conditions. It is not uncommon for user intention information to be input along with the demonstration data. [Argall, et al, 2009] This section looks at a variety of learning by demonstration policy derivation algorithms.

One approach to learning by demonstration uses Gaussian Mixture Models (GMMs) to classify the input data. [Chernova & Veloso, 2007] A GMM uses statistical probability functions to assist in classifying input data. Chernova and Veloso [2007] utilized a learning by experience demonstration technique in their approach. This means that the robot was under the complete control of the expert while experiencing the task through its own sensors. During each time step, the robot recorded its environmental state and the action selected by the human expert. Environmental state is represented using a feature vector containing both continuous and discrete values. There is a finite set of possible actions that can be selected in each training time step. The goal is to create a policy $\pi: o \rightarrow a,$ where o is the observed environmental vector and a is the selected action. A separate GMM is created for each possible action clustering all the environmental state vectors that lead to that action. [Chernova & Veloso, 2007]

During training, the observed environment data are assigned to a Gaussian mixture based upon the resulting action each time step. When enough new data points are entered into a mixture models, the model is updated and a new probability density function is generated for that action. During the learning process, the robot transitions to a state called *confident execution* as it begins to learn the policy. During this phase, the robot classifies each observed environment state and obtains a confidence value from the appropriate GMM. If the confidence value is low, which will occur when limited demonstration has occurred, the robot will request that the expert determine the action and use the new data point to update the GMM. If a high enough confidence value is found, the robot will execute the action associated with the GMM to which the input was classified. This technique was successfully applied to navigation of a circular corridor with an AIBO robot. It was also used to create a software agent representing a simulated car on a virtual road. [Chernova & Veloso, 2007] Although, this approach alone produced an adequate behavior function, there were often some problems when attempting to generalize the behavior in a new situation. To solve this, Chernova & Veloso [2007] provided additional demonstrations in problem areas. These demonstrations were used to update the behavior policy and improved the ability of the robot to generalize to new tasks.

This type of policy improvement is popular with learning from demonstration approaches. Although it is not classic learning from demonstration, the improvements found have led to development of additional algorithms that incorporate initial learning data with additional instruction. One such algorithm for policy improvement is called Binary Critiquing. [Argall, et al, 2009B] In this approach, a human teacher flags areas where the robot is performing poorly. The robot modifies the policy by penalizing the demonstration data that supported the flagged areas of poor performance. In a second similar approach known as Policy

Improvement, the human teacher correctly demonstrates the behavior that was flagged as poor. The robot updates its policy based upon the new demonstration data. [Argall, et al, 2009B]

The Binary Critiquing algorithm is designed to work on control policy that uses a scaling factor, m_i associated with each training set observation-action pair. This value is used to weight the pairs during the action selection process. When the robot performs a behavior, a human teacher identifies chunks of the trajectory as poorly executed. For each point flagged, the scaling factor associated with the chosen observation-action pair is adjusted so the pair is less attractive to the action selection process. This is very similar to the adjustment of the reward values in reinforcement learning. [Argall, et al, 2009B]

The next step in this process is the Advice-Operator Policy Improvement implementation. In this approach, rather than simply critiquing the behavior, additional demonstrations are provided. However, the new demonstration data are synthesized based on student executions and teacher advice. This approach was motivated by situations where demonstration could prove dangerous (i.e. lead to physical collision) or difficult to access (i.e. rover teleoperation on Mars). The implementation tested used a form of Locally Weighted learning. [Atkeson, et al, 1997] Both policy improvement algorithms were tested on a Segway RMP robot. In both cases, the robot was able to improve upon and exceed the performance of the demonstration teacher. [Argall, et al, 2009 B]

Another approach to policy improvement incorporates additional demonstrations with the assignment of a reputation mechanism. [da Silva, 2009] Each demonstrator is assigned a credibility rank that estimates how much this demonstrator's feedback improves the performance of the robot. The reasoning behind this ranking is that different demonstrators may have more expertise in one area over another. The demonstration in the area of expertise gets a high ranking versus an area where another demonstrator is superior. The rankings are derived from teacher critiques after the learning occurs similar to the critiquing stage of Argall, et al [2009B]. When the robot has disagreeing demonstration data from two sources, the ranking associated with the demonstrator allows the robot to choose from the more reliable source in that area. [da Silva, 2009]

A more complex task is mapping the observed state to continuous action spaces. This is known as regression. Typically, regression approaches are applied to low-level motions and not high level behaviors because the continuous-valued output often results from combining multiple demonstration set actions. The various methods can be differentiated by whether the mapping function approximation is performed at runtime or prior to runtime. Approaches that form a complete function approximation prior to run time are at one end of this spectrum. Several use neural networks as their behavior functions. Neural networks have been used to enable autonomous road driving [Pomerlau, 1991], robot arm placing of a peg in a hole [Dillmann, et al, 2005], and humanoid robot motion. [Ude, et al, 2004]

Bentivegna [2004] used a runtime regression mapping function in his learning from demonstration algorithm. His approach started with the idea that restricting behavior options by adopting behavioral primitives would speed up the process of learning the mapping. Primitives are defined as small units of behavior above the level of motor or muscle commands. These primitives were predetermined manually before beginning the learning process. When teaching a robot to play air hockey, the primitive list included straight shot, bank shot, defend goal, slow puck, and idle. Each of these primitives related to a specific action that the root was capable of executing. Before the robot can learn from observed data, a module was developed to segment the observed behavior into primitives. Once the data are segmented, the result is classified and stored in memory associated with a state vector. In air hockey, the state vector contained information about the position and velocity of the puck.

The robot was ready to play air hockey once the observed data was processed. During execution, the behavior function of the robot continuously observed the state of the game, chose the appropriate primitive to execute and executed the primitive. Choosing the primitive involved comparing the current state to the observed states in memory. If a match to current state was found, the robot agent executed the primitive chosen by the human previously observed. If no match was found, a nearest neighbor algorithm was executed to choose the observation closest to the current state. The nearest neighbor was determined using the simple distance calculation:

$$\boldsymbol{d}\,(\boldsymbol{x},\boldsymbol{q}) = \sqrt{\boldsymbol{w_j} * (\boldsymbol{x_j} - \boldsymbol{q_j})^{\boldsymbol{2}}} \qquad (2\text{-}1)$$

where x is the current state vector, q is an observed state vector and w is a vector of weighted dimensions allowing more importance to be given to various aspects of the state vectors. Once the nearest neighbor is determined, that primitive action is chosen for execution. This approach proved most successful in a virtual air hockey game where perception of puck position and velocity were near perfect in both the observed data and in actual game play. It was also used to train an actual hardware robot arm. While the training was equally successful, the robot did not play as well as its virtual counterpart due to inaccuracies in sensing puck position and velocity. [Bentivegna & Atkeson, 2001] Additional modules were added to allow the agent and robot to learn from practice as well using an approach similar to reinforcement learning. [Bentivegna, 2004] K-nearest neighbor classifiers have also been used in robotics to learn obstacle avoidance and navigation behaviors. [Saunders, et al, 2006]

2.5.7 Collaborative Learning by Observation

The one thing shared by all of these techniques is that they were developed for a single learner observing either a single expert or multiple experts performing the same event. There has been research into using multi-agent reinforcement learning for learning by observation technique. [Price & Boutilier, 2003] In this technique called implicit imitation, the learning agent called the observer is assigned another agent termed a mentor to imitate. In the implemented system, the agents are independent entities that do not consider the state or actions of the other agents in the system when determining their next action. However, their joint actions do affect the state of the overall system. Some assumptions are necessary for this type of learning. The observer and the mentor must have the same state space and must have the capability to perform similar actions. When the observer begins learning, it has no concept of the transition policy, but does have some idea of the reward functions. The observer's policy is developed by imitating the expert's actions and deriving rewards along the way. Using implicit imitation, the observing agent was able to learn a policy for traversing a two-dimensional maze previously mastered by its expert mentor. [Price & Boutilier, 2003] Unfortunately, this method is limited to examples where a similar expert agent already exists. Additionally, the application or simulation must be able to tolerate mistakes during the time the observer agent is learning. This approach is inappropriate in an environment where a mistake or error by the agent or robot could results in damage or injury.

Robotics has also addressed the concept of learning collaborative behaviors through demonstration. [Chernova & Veloso, 2008] Using a single human demonstrator, the researchers were able to teach two robots a simple sorting task. The robots were given a queue of balls in three distinct colors. Each robot had two bins for sorting the balls by color. The robots had to learn whether to place the ball in its left or right bin or pass to the other robot. In this particular example, the robots were able to query for additional demonstration when they were unsure of the next step. Support Vector Machines (SVMs) were used to classify the behaviors in this task. [Chernova & Veloso, 2008]

The existing work in learning by observation and demonstration show us that there are three clear steps to the process of learning by observation. The first step is classifying the demonstration or observation into a form understandable by the computer. The second step is to convert that information into the behavior function of the agent or robot. The final step is to generalize that behavior and determine the correct behavior for untrained steps. Often, the second and third steps are combined as done by GenCL. [Fernlund, 2007] The step most affected by training a team of agents rather than a single agent is in the first step. It is necessary to classify not only the demonstrator's current situation, but the perceived situation of the demonstrator's team in order to properly classify the current state of the team. Once that determination is made, the team situation can be treated as merely another element in the current state vector for the behavior function.

2.5.8 Learning by observation summary

Among the systems reviewed, there does not seem to be clear favorite among the types of behavior functions used. It seems as though the behavior function is designed to best fit the application or simulation being used. The final step of generalizing the behavior is necessary to increase the utility of the trained entity. Like the behavior function, the method used is highly dependent upon the classification algorithm and the application environment. Each of the methods described above, reinforcement learning, request for additional demonstration and k-nearest neighbor have all proven effective but all are not appropriate in all situations. Each method has its weaknesses. Reinforcement learning can only be used in systems where mistakes and errors do not result in damage or injury, additional demonstration is inappropriate when expert demonstrators have limited availability, and k-nearest neighbor is computationally expensive when a large state vector exists.

CHAPTER 3 PROBLEM DEFINITION

In this chapter, a detailed statement of the problem addressed by this dissertation is presented. First, the larger problem of modeling human teamwork accurately in a simulation is described. Second, the more specific problem of using learning by observation to solve this issue is analyzed along with the motivation for this approach. The last two sections discuss the hypothesis being presented and the contributions made by this dissertation to the research community.

3.1 General Problem

There are many reasons to simulate effective human teamwork. The resulting simulated team can be used as expert examples, it can provide teammates for training purposes when the whole team is not available, and it can provide realistic opponents in games and training simulations. Successful teamwork simulations exist, but all require that the behavior of the team members be programmed into the simulation. Typically the method of programming requires at least some specialized skills. Programming could be in a scripting language, XML or traditional high-level computer software code. None are able to observe a team at work and replicate the teamwork behaviors. Several multi-agent systems devoted to the development of simulated teams exist that require time consuming knowledge acquisition and programming skill in order to adapt them for different domains.

Learning skills by observation has been successfully performed for individual entity simulations, but effective teams require that the individuals have task-specific skills and teamwork skills. When team members are trained individually and given a common goal, the results are often merely adequate. However, most sports fans will tell you that an adequate team does not usually have a winning season. Playing games against an adequate opponent will quickly become boring and training will not be particularly productive. An effective algorithm for learning by observation needs to include the capability to learn those extra things that make a team greater than the sum of its individual parts.

3.2 Specific Problem

There has been no algorithm for learning effective collaborative behavior from observation. While it is possible to train each team member individually in their portion of the task using observational learning, this type of training does not currently capture the collaborative behavior of the team and results in a less effective team. A system capable of quickly and easily capturing both collaborative behavior and domain specific behavior in a single step would greatly simplify, improve the fidelity of, and accelerate the development of simulated action teams. The use of simulated teams, particularly in entertainment and training applications, is growing steadily as evidenced by the growing number of multi-player games and computer-based training programs available for use. An algorithm that can accelerate the creation of these simulated teams when expert performance is available to observe and emulate would decrease cost and increase productivity of the creators of the simulations. Developing such an algorithm is the specific problem addressed by this dissertation.

3.3 Motivation

The knowledge acquisition required to recreate an effective team in this manner is time and skill intensive. Experts in teamwork skills are needed as well as individuals capable of capturing their knowledge and translating it into computer code or scripts for the simulation to use. These teamwork experts are often few in number and in high demand, so the ability to acquire knowledge from them is limited. Acquiring knowledge from an expert verbally can also inadvertently result in the loss of implicit or tacit knowledge. An algorithm capable of learning by observation of a real or simulated expert team would be a significant step forward in the quest to create simulated teams quickly, easily and of high fidelity.

3.4 Hypothesis

Several collaborative multi-agent systems currently exist. There has also been success in training simulated entities by observation of expert performance. The hypothesis of this dissertation is that the current research in these two areas can be combined to take a multi-agent system and train it to effectively simulate the actions of an expert action team.

3.5 Contributions

- Creation of a novel approach to training simulated action team via observation.
- A unique multi-agent system for the creation of simulated action teams from observation of expert team performances.
- Evaluation of various learning by observation techniques previously developed for single agent training for use in a multi-agent system.
- Application of this approach in a variety of simulated action teams with increasing complexity.
- Prototype collaborative multi-agent learning system available for use by other researchers.
- Data from prototype system evaluation.

CHAPTER 4 CONCEPTUAL APPROACH

This chapter describes an approach to learning collaborative behavior by observation. The goal is to duplicate or imitate the behavior of a team autonomously, without actual interaction with the observed team. The concept behind this new approach to learn team behaviors is that the actions leading to effective teamwork require constant coordination among the team members. In order to build a model of coordinated action among team members, it is important that the members of the team build an implementation of a shared mental model. In a simulated entity, this shared mental model is implemented as shared or collaborative data. While this shared data need not be identical for each team member, it does require that they be aware of each others' movements and actions. This collaborative data builds the foundation for the implementation of the shared mental model.

Observational learning techniques allow for the learning of implicit behavior as well as explicit behavior. The combination of observational learning with the inclusion of collaborative data will facilitate the learning of tacit understandings and interactions among the teammates; these are collaborative behaviors that the team members may not even be aware that they are performing. This moves beyond the Joint Intention Theory that states that they only need to share the intention, and not knowledge about each other’s actions in order to work together effectively. So while each team member is trained individually, the training data used includes the usual information about individual tasks and adds information on teammates actions and intentions.

This, however, gives rise to a problem of potentially too much data – much more than can be handled efficiently by a single-entity learning mechanism. To address this, the approach decomposes the behaviors into contexts. Each agent identifies its individual context by learning cues about the environment. Once the context is identified, the potentially overwhelming amount of data can be reduced to include only the data relevant in that particular context. The list of applicable behaviors is also limited to those appropriate for the context. These reductions turn the overwhelming problem into a solvable one.

A system was developed named **C**ontextually-based **O**bservational **L**earning of **T**eamwork **S**ystem (COLTS). This system is a set of tools and processes that can be combined to create a simulated team from observation. The tools include a Collaborative Context-based Reasoning (CCxBR) multi-agent framework and a learning algorithm derived from single entity learning by observation systems. The idea behind COLTS is to create a generic approach to learning coordinated team behavior by observation. While some adaptations must be made for various application domains, the bulk of the system is reusable from application to application. The tools are divided into three parts: observation, off-line learning and run-time multi-agent framework. The output of the observation feeds the learning portion. The output of the learning portion is then used by the run-time multi-agent framework to drive the behavior of a team of agents. These agents emulate not only the individual expertise of the observed team members, but collaborative behavior as well. To demonstrate this flexibility and adequately evaluate the system, a total of three separate prototypes in different applications were developed, each in a different application. These are defined at the end of this chapter and discussed in detail in subsequent chapters.

4.1 Conceptual Approach

COLTS approaches the problem of learning collaborative behavior from observation in a manner very similar to previous single-entity learning from observation approaches. However, there are several key enhancements needed to capture the collaborative elements of the team and scale upwards from a single entity to an entire team. The hypothesis presented in Chapter 3 states that the combination of multi-agent framework proven to have the elements necessary to perform collaborative behavior with a proven learning from observation algorithm are the answer to observing and emulating collaborative behavior.

As the chosen multi-agent framework, CCxBR has been proven to have the ability to perform collaborative behavior. [Barrett, 2007] CCxBR links the agents representing the teammates together through the use of a team context. The team context is designed to track and share the individual contexts of all the team members. Knowing the context of teammates gives each team member agent valuable insight into the state of the team and serves as important collaborative data. This collaborative data can be included into the individual situational and task data of each team member and used to drive the behavior function of the team members.

One of the largest drawbacks to learning the behavior of an entire team rather than a single entity is the large amount of data needed to represent this collaborative information at both observation and run-time. A software agent working alone has to obtain and use data related to the task it is undertaking. A software agent working as a team member must get that data as well as collaborative information about the state of the team goal and its teammates. COLTS and CCxBR address the handling of this large amount of data through the use of context. In psychology, context has many definitions mostly related to the concepts of situation, background, milieu and field. [Bazire & Brezillon, 2005] In COLTS, context is defined by the values of the data describing the current situation of a particular agent. The use of context allows the problem of driving imitative behavior to be broken down into smaller problems where the amount of data used in any given context is limited to a more manageable amount.

The use of CCxBR as the run-time portion of COLTs did not force any decisions on behavior functions or learning algorithms. However, it was important that the method chosen be capable of capturing and using the collaborative information provided by the team context at run-time. The run-time behavior function has its roots in robotic memory-based policy and case-based reasoning. The function is driven by a behavior map that contains a set of situations or cases against which the current situation of the agent can be compared. When an exact or similar situation is found in the behavior map, the agent can extract the action associated with that situation from the behavior map and use it to drive its next step. The primary difference between this approach versus the inspiration approaches is the inclusion of collaborative data in both the situations stored in the behavior map and the run-time situation.

The COLTS learning algorithm is described in detail in Section 4.4.2. It builds these behavior maps from observed data. It takes inspiration from single-entity learning from observation and adds the ability to collect and use collaborative information. This can be done because the observational module does not collect information one team member at a time, but rather collects team information at each time increment available. Having this collaborative data available during learning allows for it to be factored into the learned behavior function.   So at run-time, the agent gathers its task data from its sensors and data source and collaborative data from the team context and compares both against the stored situations in the behavior map.

The use of context is also very important in making this approach scalable for use in teamwork as well as single agents. A separate behavior map is developed for each individual context. Without the breakdown into contexts, it is quite likely that the behavior maps would contain a very large number of entries and make the run-time process of finding a match take a large amount of execution time.

The following sections describe in more detail the implementation and approach of the COLTS system. The system is broken down into three modules: observer, learner and run-time. The ability to learn and perform effective teamwork using COLTS hinges upon the ability of each team member to observe, learn and use collaborative data at run-time in addition to individual task data. COLTS provides the tools and algorithms necessary to make possible the inclusion of collaborative data through context and behavior maps.

4.2 Learning Collaborative Behavior by Observation

The approach to learning collaborative behavior by observation has many similarities to learning by observation for a single entity. The idea is to replicate or imitate the behavior of an example entity or entities while performing a task. Both single-entity and collaborative systems require the acquisition of observed data for use in training and testing the system. The difference between single-entity and collaborative behavior in the data acquisition phase is primarily the amount of data needed. Compared to a single-agent learning-by-observation algorithm, the amount of data needed to recreate behavior is typically multiplied by the number of agents being observed. Furthermore, depending upon the application, there can be additional collaborative data. Depending upon the team activity, collaborative data may be as simple as messages being passed among the teammates or can encompass detailed information about teammates' position and activities. In CCxBR, collaborative data typically include the team context and information about the current context of the teammates.

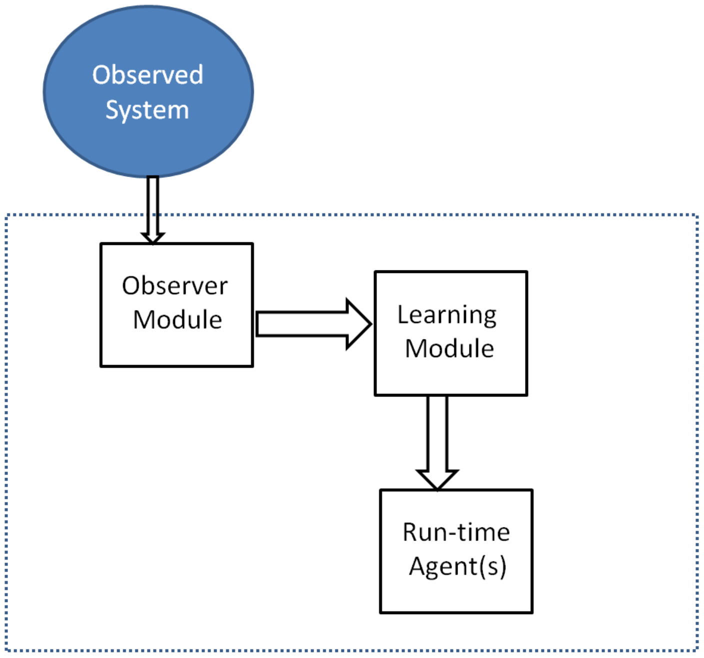


Figure 4-1 Typical Learning by Observation System

Learning-by-observation systems commonly break the task down into modules. Typically, there is an observer module, a learning module, and a run-time module. Figure 4-1 shows a block diagram of the usual components. The observer module is responsible for acquiring the observed data from the entity or entities being learned from. It is responsible for taking the observations and converting them into a format readable by the learning module. The observer module used in this dissertation consists of data logs output by a simulation. These data logs exhibit the knowledge (albeit indirectly), movement, and actions of the various team members. The simulation is agnostic as to the source of the team members' behavior. The observer module is incorporated in the observed system in such a way that it does not interfere with the behavior and timing of the observed agents in any way. These agents can be programmed software agents, human performers, or some combination of the two.

The observer module can be one of the most difficult aspects of a learning-by-observation system to implement, particularly when attempting to observe behavior of physical (i.e., non-simulated) entities. In some cases it is possible to instrument physical entities to provide position and action data electronically. This approach was used by Fernlund, et al [2009] where they were able to observe tank movements via instrumentation on actual tanks. The tank instrumentation transmitted GPS data on tank movements and actions to a central server where it was placed into logs similar to those used in this dissertation. If such instrumentation is not available, it is sometimes possible to use image recognition software to create logs from video capture. This approach was used by Bentivegna [2004] to capture human behavior for robotic learning by demonstrations.

The second phase of learning by observation is processing the collected data and learning from it by using a learning module. This involves translating the observed data into some learned function usable by the software agent or agents replicating the behavior of the observed entity or team. In the case of COLTS, the learning module must translate the simulation data logs into a behavior function usable by the run-time agents. Several learning by observation techniques were discussed in Chapter 2. The technique used in COLTS is a memory-based policy technique inspired by the learning by demonstration work of Bentivegna [2004] and the case-based reasoning techniques used by Floyd, et al. [2008]

The final step is the recreation of the original behavior in a run-time agent or agents. The approach used here is by necessity tied very closely to the learning algorithm. The output of the training algorithm is used to drive the behavior of the new agent. The choice of CCxBR as the paradigm used to implement the new simulated team does not dictate a particular behavior function. This opens the door to a variety of possibilities. Previous implementations of CCxBR simply used a manual approach to create the behavior function based on input from domain experts. There was no obvious choice of a behavior function for use in capturing collaborative behavior. Ideally, the chosen technique for capturing the behavior would scale linearly, meaning that the amount of work to scale upwards would correspond directly to the number of teammates.

Since CCxBR consists of CxBR agents, the next logical place to investigate was the implementations used in previous CxBR learning by observation systems. Specifically, FAMTILE [Stensrud & Gonzalez, 2008] and GenCL [Fernlund, 2004] were analyzed as potential methods for use. While successful for single agent learning, the complexity of the two systems would most likely prove very difficult to scale linearly into a teamwork application. The robotic learning by demonstration approaches reviewed in Chapter 2 showed more promise for linear scalability. The memory-based approach to behavior used by Bentivegna [2004] is the inspiration for the learning algorithm of COLTS. A memory-based policy will need to be developed for each team member and possibly for the team context transitions. This amount of work is close to a linear scale. In addition to Bentivegna's successful research in robotics, a very similar case-based reasoning technique was successfully used to learn simulated soccer techniques by observation. [Floyd, et al, 2008] Floyd, et al [2008] developed a single case base for a single entity. When applied to a group of entities, the number of additional case bases would be proportional to the number of entities.

Bentivegna [2004] created a system that taught a robotic arm to play air hockey by observing a human playing the game. He began by using human domain knowledge to create a series of primitives which translated into game actions. These were basic game skills such as defend_goal and straight_shot. When a predefined critical event such as the puck approaching the paddle was observed, a state vector was created. This vector contained information about the puck, the paddle and the primitive being executed. The recorded state was associated with the primitive action taken and stored into a memory-based policy. During run-time, the robot observes the current state of the game, compares it against the stored state vectors, and executes the action associated with the stored state vector that is closest to the current state. A k-nearest neighbor algorithm is used to determine the closest state vector. Initial testing was done with a simulated arm and game, but eventually transferred successfully to a physical robotic arm and actual air hockey table. The method was further tested by teaching the arm to roll a marble through a wooden maze. [Bentivegna, 2004]

Floyd, et al [2008] used case-based reasoning as the behavior function for their software agents that learn to imitate from other agents. The observation and learning modules of their approach create a case base consisting of example states with actions taken. The case base serves the same function as the memory-based policy used by Bentivegna. At run-time, a k-nearest neighbor algorithm is also used to determine the case to use from the case base. Rather than robots, Floyd, et al train simulated soccer agents based upon expert examples provided by winners of the annual RoboCup simulated soccer league. The technique has proved successful at training single players to imitate other players performing a limited number of basic behaviors such as Chase Ball and LineRun. [Floyd, et al, 2008]

4.3 Behavior Maps

Bentivegna's memory-based policy and Floyd's case-based reasoning approach were the inspiration for the behavior function at the center of the COLTS toolkit. *Behavior maps* drive the behavior of the COLTS team. A behavior map is analogous to the case base in case-based reasoning. The behavior map is differentiated from these approaches by the inclusion of collaborative data in the state vector. The inclusion of collaborative data enhances the traditional approaches for use in emulating an entire team. Additionally, both of the fundamental approaches used a single policy or case base for each entity. Since CCxBR breaks the behavior of the team and team members into contexts, there will be a behavior map for each identified context in an application. This is vital to creating a scalable application. Limiting the number of state vectors to compare against to those relevant to a particular context keeps the size of the behavior map small enough to execute more quickly than a single case base or policy.

The behavior of the observed agent or agents can be thought of as a series of situations and actions. The situation can be represented by a state vector S containing the available information about each agent's environment. For use in learning collaborative behavior, this vector must be expanded to include information about teammates and team goals. The behavior maps are built from observed data that contains both environment information and collaborative information. The situation vector S will contain m items of environment information, E, and n items of collaborative information, C. At run-time, the environment information is derived from agent sensors and communications and the collaborative information is provided from the team context. The values of m and n will vary based upon the application domain. The actual collaborative information stored in S may be specific contexts of teammates or information about the teammate such as position and bearing from which the teammates' context can be determined. It is also important to determine if all teammates' contexts are relevant and available at run-time. In many team situations, only certain teammates' information is relevant to determining behavior.

$$\boldsymbol{S = \{E_1, E_2, .. E_m, C_1, C_2, .. C_n\}} \qquad \textbf{(4-1)}$$

There are a finite number of possible situations for each domain. The number of possible situations is driven by the possible values of the data elements contained in S. It is usually not necessary to know all possible situations before building the behavior maps.

$$\boldsymbol{PossibleSituations = \{S_1, S_2, \ldots., S_n\}} \qquad \textbf{(4-2)}$$

Each agent also has a set of possible actions A that can be taken. This list is typically driven by the simulation environment. Most simulations have a limited number of actions available for agents. These available actions are the members of the possible actions vector. This is a finite list of 1 to m possible actions. However, a particular action may have a parameter associated with it. For example, a MOVE command may have a parameter indicating the heading in which to move.

$$\boldsymbol{PossibleActions = \{A_1, A_2, \ldots., A_m\}} \qquad \textbf{(4-3)}$$

The actual elements of the Situation and Action vectors will vary from application to application, but each application will have a definable situation and a finite set of actions. In any given simulation, the behavior of the agent can be seen as a series of Situations, $S_{x,,}$ followed by an Action, A_x that leads to a new situation S_{x+1}. Because this mapping contains more information than a typical case base or memory-based policy, this dissertation defines the mapping of situations to actions as a *behavior map*.

$$\boldsymbol{Behavior\ Map = \{\{S_4, A_2\}, \{S_6, A_1\}, \{S_{25}, A_3\}, \ldots\}} \qquad \textbf{(4-4)}$$

The COLTS learning module is designed to build the behavior map based upon observed data from the expert team. The run-time agents use the behavior maps to determine the actions of the run-time agents.

A teamwork application is very likely to have a larger situation vector and choice of actions than the robotic arm [Bentivegna, et al, 2006] and single soccer player [Floyd & Esfandiari, 2008] implemented using this method. Without modification, this approach can consume large amounts of memory to store the mappings of situation and action and the k-nearest neighbor algorithm could potentially consume excessive processing time. However, COLTS uses collaborative context-based reasoning (CCxBR) to break the problem down using the natural contextual partitioning of the problem. This breakdown of the problem into contexts enables the development of a separate behavior map for each context making the use of behavior maps feasible by limiting the size of the behavior maps. The size of each behavior map is determined by the number of mappings of situations to actions stored in the map. The use of a single behavior map using this approach for a team of agents would make it a large consumer of memory and probably too slow for practical use. The use of contextually-based behavior maps will also address the possibility that the expert might encounter very similar situations in different contexts that result in differing actions.

4.4 Scalability of the COLTS Algorithm

The scalability of behavior maps is directly related to the size of the situation vector, S. The size of S would differ based upon the application and data available to the entity, but will be a fixed size for each application. While each context may not use all of the data within S, all data in S is available for each context. For definition purposes, the situation vector, S, requires *n* bytes to implement. The size of the action vector, A, will also vary from application to application but can be defined as *m* bytes. If behavior maps were implemented for a single entity, there would be *j* contexts. This would result in *j* behavior maps. The size of each behavior map, *b*, can be represented as a sum of the size of one situation vector plus one action vector times the number of entries in the behavior map.

$$\boldsymbol{b = (n + m) * number\ of\ entries} \tag{4-5}$$

The number of entries in the behavior map is driven by the training data available. Each behavior map will have a different number of entries based on how often the observed entity enters a particular context. Ideally, a similar number of entries for each context will be found and the resulting total size of all behavior maps will be the number of contexts times the maximum size of a behavior map.

$$\boldsymbol{memory = b * j} \tag{4-6}$$

Execution time for behavior maps is also related to the size of the situation vector and the number of entries in the behavior map. At run-time, the worst case is that the k-nearest neighbor algorithm must compare all of the entries in the behavior map against the current situation. If we calculate that this algorithm requires q seconds for each data element in the situation vector, we can calculate the worst case run-time, *r*, for a single-entity as follows:

$$\boldsymbol{r = q * n * number\ of\ entries} \tag{4-7}$$

So, the two biggest factors in determining the run-time viability of a software agent using behavior maps to drive the behavior function are the number of data elements in the situation vector and the number of entries in the behavior map.

When scaling upwards for a team as opposed to a single-entity, it is possible to calculate the worst-case impact of teamwork on the number of data elements. As stated above, the situation vector consists of two types of data; environmental data elements, E, and collaborative data elements, C. The size of a team does not typically impact the number of environmental data elements for an agent, but it does impact the number of the collaborative data elements. In a worst case scenario, the amount of collaborative data would be multiplied by the size of the team. If a situation vector has *v* environmental elements and *w* collaborative elements for each team member and a size of *u* team members, the size of the situation vector becomes:

$$\boldsymbol{n} = \boldsymbol{v} + (\boldsymbol{u} * \boldsymbol{w}) \qquad \textbf{(4-8)}$$

If the contexts for a team application are appropriately chosen, the number of entries in each behavior map should remain about the same as with a single element. However, in a worst-case scenario the number of total contexts could also be multiplied by the number of team members, u. This means that the total memory usage of the behavior maps is multiplied by the number of team members, u. Since execution time is impacted by number of entries and situation vector size, *n*, the worst case execution time for an individual agent is also the single entity time multiplied by the team size, u. In big-O notation, the scalability of both memory and execution time would be O(u) where u is the size of the team. This makes the COLTS algorithm linear in complexity

4.5 COLTS Toolkit

The COLTS toolkit is a collection of processes and tools that combine to form a learning by observation toolkit. It is intended to be a generic starting point capable of being adapted to observe and imitate the behavior of different types of teams. It is intended to reduce the amount of time and work necessary to create a simulated team. Learning from observation systems have been proven to reduce the time and effort necessary to acquire the knowledge to create an expert agent. [Sidani, 1994] [Stein, 2009] [Fernlund, 2004] [Bentivegna, 2004] [Floyd, et al, 2008] So it is expected that it will provides the same reduction for developing a team of expert agents. In addition, the run-time framework provides reusable agent and context classes, which are designed to reduce the amount of software development. This dissertation presents a basic system capable of adaptation to various teams. In addition to the various tools, a set of processes for adaptation to various domains is provided. The system is flexible enough to support the future addition of tools to further automate and simplify the process of creating a simulated team with behavior based upon an observed team. A conceptual description of each module within the COLTS toolkits and the associated processes are described in the following sections.

4.5.1 Observer Module

In order to learn by observation, there must be some form of observation. In the case of COLTS, it is necessary to observe the current situation vector at time (t, S_t), and note the action taken within that time-step, *A*. In the prototypes developed in this dissertation, all the teams being observed are part of a simulated environment that creates log files consisting of situational data and action taken by various members of the team. A set of data about each agent is logged at the end of each simulation time step. Each simulation log file had a different format and different data.   The log files serve as input to the COLTS learning module. Each of the three prototypes developed for this dissertation used a slightly different approach to the observer module and the log files all had different formats. However, all three used log files automatically generated from the simulation in which the observed team operated. A possible extension to COLTS could be tools capable of generating similar textual log files from images or binary data when the observed team is not operating in an environment capable of creating usable log files.

The observer portion of COLTS is the least reusable and most domain-specific of the three portions of the system. This is the step requiring the most domain knowledge and expertise. If access to a small amount of time with a domain expert is available, this is the portion of the process where it would be used most effectively.

First, a team to observe must be chosen and log files must be collected for use as training data. The choice of the observed team will probably be influenced by the success and expertise of the team, but the criteria should be determined by the end use of the COLTS team. If the observed team operates in a virtual environment such as a simulator or computer-based game, the observation module must do less work to translate the team's situation data and actions into training data for the learning module. All the teams observed for the prototypes developed in this dissertation operated in a virtual environment, where a log file containing at least some information was already being produced. Some domain knowledge of the virtual simulation was needed in order to determine whether the content and type of data being logged was sufficient to serve as training data. In the third prototype developed, additional pieces of data were added to the existing logged data in order to create adequate training data. This additional data logging was added only to ease the development of the third prototype and was not part of the original simulation. In all of the prototypes, the logging was done by the simulation and not the software agents representing the observed team.

The next step is analysis of the data available in the log files. The expected outcome of this step is a definition and software implementation of the data elements used to create the situation vector, S and an enumeration of the possible actions for the team. The format of this data structure requires knowledge of the environment in which the COLTS team will operate. The situation vector data will consist of the information available to each team member at run-time. If the team under development is the cockpit crew of an aircraft, the situation vector data would contain the information presented to the crew by the flight instruments such as altitude, air speed, landing gear status and so on. It should also contain collaborative information, such as current team context, teammate contexts and communication from team members. It is also important that this type of data be available in the training data log files from the observed team.

The next set of data needed is the list of possible actions. These data represent the action vector, A. Typically, this list is driven by the type of actions available in the simulated environment. In the case of the cockpit crew, this list might contain such things as adjusting airspeed, adjusting flaps, lowering landing gear and turning the aircraft. It is not always necessary for the action taken in any given time-step to be explicitly logged. If the action can be derived from the change in situational data between two time-steps, that is sufficient for training. For example, if logged data indicates that landing gear is up during time-step t, but time-step t+1 indicates that it is now down, it is fair to infer that the action to lower landing gear was taken in time-step, t. Once all possible actions are known and enumerated, the building blocks of the behavior maps became available. The situation vector S is defined and ready to be populated with values and the possible actions are enumerated and ready to be paired with populated situation vectors.

The final step of the observation stage is contextual analysis. Each problem domain will have a unique set of contexts. There has been some research into automatically determining contexts for a single agent in a particular problem domain. [Trinh, 2009] However, this has not been expanded into determining contexts for a team of agents. Therefore, it is necessary for the researcher to analyze the behavior of each observed team and manually determine the contexts used. If domain expertise is available, this is the step where it should be used. If no domain expertise is available, it is still possible to continue with contextualization, but the resulting contexts may be suboptimal. An optimal contextualization will result in fairly evenly sized behavior maps and better emulation of observed behavior. At the worst, it may not be possible to create a set of meaningful contexts without some domain expertise. The team contexts should be determined first. These contexts represent the current goal of the team as they proceed through the mission. Continuing the example of a cockpit crew of a commercial airliner, possible team contexts might include take-off, landing, ascent, descent and cruising. The crew of a different type of aircraft or mission such as one that involved search and rescue might have additional contexts (e.g. search patterns). Once a preliminary list of contexts is manually drawn up, it is necessary then to look at the situation vector and see which data indicate a particular context. For example, the landing gear being down would indicate that the aircraft is probably either in a take-off, a landing or a ground operation context. Additional data such as speed, altitude and aircraft angle of attack could be used to further narrow down the context of the crew. If there is insufficient data in the situation vector to differentiate between two contexts, it may be necessary to combine those two contexts into a single context. Once the team contexts are determined, each team member must be analyzed to determine their individual context. Like the team context, domain knowledge should be used to list possible contexts and available data examined to ensure that the contexts can be distinguished using the available training data. Some overlap of individual context is expected between the team members, so the final step after each team member is analyzed is a consolidation of all the individual contexts into one list.

Context specification is very highly tied to the application domain. It does require domain knowledge and expertise. The COLTS prototypes developed for this dissertation each had very different contexts and very different factors driving the contexts. More details about the contexts chosen for each of the prototypes are provided in Chapters 5, 6, and 7.

4.5.2 Learning Module

The learning algorithm portion of COLTS is responsible for processing the observed training data and building contextually-based behavior maps. It takes as input the outputs of the observation module. This includes the training data log, the situation vector data structure, the enumerated list of actions, and list of team and individual contexts. The list of contexts should include the criteria for determining that context from the available situational data. The learning module implements an algorithm that accepts these inputs and creates a set of behavior maps for use by the COLTS run-time agents.

In order to derive the behavior map for each individual context, the entire training log file must be read and parsed into situation vectors. The expectation is that the log file data is ordered sequentially by time. This means that the data for all team members are grouped by time-step. A typical log entry would contain the simulation time, information for team member #1, information for team member # 2, and so on until information for all team members is presented. The entry may also contain state data not associated with a particular team member, such as the current score in a simulated game. This type of data represents collaborative data in the situation vector and typically used by all members of the team.

Figure 4-2 describes the algorithm used by COLTS. It begins by creating empty behavior maps for each context. The number of team members in the observed team determines the number of times the log file is parsed through. For each team member, the log file is reset to the beginning. An application-specific function is called that will parse a time-step from the log and place the data into a Situation class called *previousSituation*. Because each time-step contains information about all team members, this function accepts a team member designation as input to determine how to parse the information appropriately. Additionally, any actions logged in that time-step for the designated team member are placed into a data structure called *previousActions*. The context of the designated team member in the parsed time-step is determined by another application-specific function called *getContext*. This value is stored in *previousContext*. The algorithm next enters a loop until it hits the end of the file. The application-specific parsing is called again to create a Situation class called *currentSituation* and an action class called *currentActions*. The context of this Situation class is determined by *getContext* and placed in a variable called *currentContext.*

The next step is comparison of previousSituation and currentSituation. This comparison will determine if any non-logged actions such as a context-transfer occurred between the two time-steps. If so, these actions are added to the Action class called *previousAction*s. Finally, behavior map for *previousContext* is retrieved. The information in *previousSituation* and *previousAction* are paired and added to the retrieved behavior map. Finally, before the next time-step is parsed, currentSituation and currentActions are saved as *previousSituation* and *previousActions.*

When the last time-step has been parsed for the last team member, the behavior maps are saved to files. Each behavior map is saved in an individual file named for its associated context. For example, the behavior map for a LANDING context would be named "LANDING.MAP".

```
Learning Algorithm
Inputs: log file,  context names, size of team (u)
Outputs: a behavior map for each defined context
Train()
{
    open log file;
    // parse log file for each team member
     for ( i = 1 to u)
      {
        reset log file to beginning;
         // parse situation data for timestep from log as
         // related to agent i
        previousSituation = parseTimeStep(i);
        // parse any actions defined in log
         previousActions = parseTimeStepActions();
        // determine context based on situation data
        previousContext = getContext(previousSituation);
        while (another time step exists)
         {
            // parse situation data for timestep from log as
            // related to agent i
            currentSituation = parseTimeStep(i);
                  // parse any actions defined in log
            currentActions = parseTimeStepActions(i);
            currentContext = getContext(currentSituation);
             // if previous and current context are different
             // add a context change action to action list
                   if (currentContext != previousContext)
                          previousActions + = getContextChange(
                         previousContext, currentContext);

             // get behavior map for previousContext
             bMap = getBehaviorMap(previousContext);
             // add mapping of previousSituation and
             //  previous actions
             bMap.add (previousSituation, previousActions);
             // get ready for next timestep
             previousSituation = currentSituation;
             previousActions = currentActions;
          } // end while loop
        } // end for loop

        for ( each context)
        {
           bMap = getBehaviorMap(context);
           save bMap to context.map;
         }
} // end Trainer
```

Figure 4-2 Training Algorithm Pseudo-code

4.5.3 Run-time Module

The COLTS run-time module consists of a generic multi-agent framework capable of being customized for each application and a contextually weighted k-nearest neighbor technique. These provide the elements of the system that recreate the behavior of the observed team as COLTS agents.

4.5.3.1 Agent Framework

The choice of a multi-agent framework was primarily driven by the research into the psychological basis of human collaborative behavior, as discussed in Chapters 1 and 2. A framework capable defining team goals and implementing shared memory models is a necessary part of COLTS. The framework must be capable of accurately representing human behavior in addition to team goals and shared memory models. The Collaborative Context-based Reasoning (CCxBR) paradigm [Barrett, 2007] was chosen as the basis of COLTS run-time agent framework because it demonstrates all of these characteristics and was the clear choice for the multi-agent framework portion of COLTS. Each agent in the framework is a CxBR agent [Gonzalez, et al, 2008], a proven paradigm for representing human behavior. The addition of a team context in CCxBR provides a mechanism able to capture collaborative behavior and implement a representation of shared memory model. In addition, CCxBR has also been formally shown to effectively model collaborative behaviors as defined by Joint Intention Theory, an important element in reproducing collaborative behavior. [Barrett, 2007]

The CCxBR framework contains a set of generic software classes that accept behavior maps as input to create the behavior functions for a team of software agents. These agents are linked together by a team context. At any given time, each agent is in a particular individual context. In COLTS, the contextually-based behavior maps developed by the training algorithm are used to develop the behavior function. Each context has a unique behavior map. Each agent will begin a time-step by building a situation vector, S, using information available to the agent from sensors or messages. S will be compared against the situation vectors stored in the current context's behavior map. A contextually based k-nearest-neighbor algorithm described in the next section will be used to determine the stored situation most similar to S. The agent will then execute the actions paired with the stored situation. This allows for a significant amount of software reuse, as each context and agent within a particular application inherits from the same software class. The behavior map provides the uniqueness to each context.

4.5.3.2 Contextually weighted k-nearest neighbor technique

The behavior maps can only contain the states encountered in the log files used for training. Typically, this is a subset of all the possible situations. So, there must be some method to determine actions when the current situation faced has not been stored in the behavior map during run-time. The most direct solution is to find the situation "closest" or most similar to the current situation. Defining similarity among the various situation vectors is basically a clustering problem. Because the calculations may be done during a time-sensitive simulation, it is important to choose a clustering algorithm that can be executed quickly. It must also be generic enough to be used in a variety of applications. Bentivegna [2004] used a nearest neighbor algorithm in his robotic arm applications. The calculation will return *the* state vectors that is defined as "closest" to the original vector x. The equation used there was :

$$\boldsymbol{d}\,(\boldsymbol{x},\boldsymbol{q}) = \sqrt{\boldsymbol{w_j} * (\boldsymbol{x_j} - \boldsymbol{q_j})^2} \qquad \textbf{(4-9)}$$

where x and q are state vectors being compared and w is a vector of weights. The weight vector allows for different elements to have more or less importance in the calculation. For example, in a sports team application, a difference in position of the ball might have more importance than the difference in position of a teammate. In this equation, differences are squared to eliminate possible negative differences, then multiplied by the weight and finally, the square root is taken. This nearest neighbor algorithm is a calculation of the Minkowski distance with n =2. [Xu & Wunsch, 2005] The Minkowski distance is a commonly used clustering algorithm for finding similarity objects in Euclidean space. [Xu & Wunsch, 2005]

Floyd, et al [2008] used a similar k-nearest-neighbor algorithm to implement the distance calculations for his simulated soccer player. The version used was city-block distance which is a specialized version of Minkowski distance with n=1. [Xu & Wunsch, 2008] In city-block distance, the absolute value of the difference is used rather than the square. This is typically used in applications where the distance is measured in two-dimensions. The name city-block distance refers to a typical calculation of distance on the streets of Manhattan where the streets are laid out in a grid fashion. This change should shorten the execution time of the distance calculations without impacting the effectiveness of the calculations. The city-block distance calculation was the one chosen for use in COLTS. The equation is:

$$\boldsymbol{d(x,q) = \sum w_{cj} * abs(x_j - q_j)} \qquad \textbf{(4-10)}$$

In the above equation, x and q represent the situation vectors being compared and w represents the weight vector. One of the benefits of using a contextually-based framework is that the set of weights used can be different for each context. This is particularly useful when two team members have drastically different tasks to perform, but may encounter similar situations. The initial weights chosen are typically an estimation based upon knowledge of typical behavior in the context. It is expected that not all situation vector elements will be relevant to each context and by setting the weights of irrelevant elements to 0, they can be eliminated from the calculation. Relevant elements can be given an initial weight of 1.0 in each context or if some knowledge of the application exists, educated guesses for the weights can be used. For each application, it is expected that some trial and error experimentation with different weight values on relevant elements becomes necessary to find the optimal weights for each context.

4.6 Theoretical Example

Let's look a simple example of how COLTS could be used to learn behavior from simulated entities. Using the TeamBots software [Balch, et al, 2000], the user can create a variety of scenarios where simulated robots move through a virtual landscape. One such application is a simple game of "catch" between two simulated robots. The two robots alternately kick the ball back and forth across the soccer field. There is no real goal to the game; it is simply an exercise in moving the robots towards the ball and kicking at the appropriate time. In this simple team application, there is the need for collaborative behavior between the two robots. The collaborative behavior involves knowing which robot's turn it is to kick the ball. Without knowing this, the robots would simply move around the field trying to kick the ball all the time rather than taking turns. The TeamBots simulation can be instrumented to output player positions, ball position and information about turns to a data log. This section discusses the steps necessary in each of the three COLTS modules to create agents able to create new COLTS agents that reproduce the observed robots behavior.

4.6.1 Observer Module

The output of the observer module is a textually-based training log file, defined situation vector, defined action enumerations, and a list of contexts and their criteria. The TeamBots-generated log file is ordered sequentially. A time-step within the TeamBots simulation is defined as 50 ms. At each 50 ms time-step, the simulation will first log the position, steer heading and speed of each robot. Steer heading is defined as the angular heading of the simulated robot. In addition, the position of the ball is recorded and a notation of which robot's turn it is to approach and kick the ball back towards the other robot.

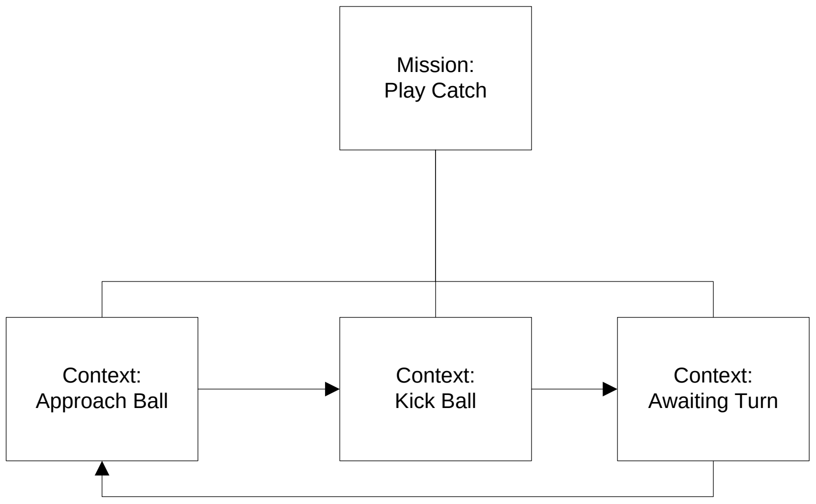


Figure 4-3 "Catch" Player Contexts

Once the data log is obtained, the next step in the COLTS process is to manually identify the contexts used by the robots. Since this is a simple example, there are not many contexts involved. Analysis of the task shows the contexts shown in Figure 4-3. The figure shows that there are three player contexts: *Approach Ball*, *Kick Ball* and *Awaiting Turn*. The shared Team Context will have two possible values: *East Player Turn* and *West Player Turn*. Once the context is determined, the next step is to identify the elements of the situation vector S as well as the possible actions A taken by the robots. The elements of S must be data that are available to the robots at run-time as well as available in the data logs. The first data element of S will be the current individual context and the second will be the team context. These elements are available at runtime and easily derived from the logged data. At runtime, each robot is able to obtain its own position in x, y coordinates with 0,0 being the center of the field. In addition, they are able to receive the position of their opponent and the ball in relation to their own position. The opponent relational data are in the form of a data structure containing the difference in x, difference in y, distance and angle. Although the logged data of opponents are in absolute x,y, it is easy to derive the relational data from the log file. The ball position is not noted explicitly, but rather as an angle to and distance from the player. The team and individual context for the opponent can also be derived. The team context is derived from the data logged about which robot's turn it is to kick the ball. The team context as well as position data also drive the determination of individual context. Table 4-1 shows a list of the data elements in the situation vector S. The possible actions that each robot can take are listed in Table 4-2. The actions include the possible movements of the robot. Additionally, it includes individual and team context transitions. Although the initial team and individual contexts are set at the initialization of the simulation, any transition after that is made by the behavior function of the run-time agents as driven by the behavior maps.

Table 4-1 Data Element in Situation Vector

Name	Data Type	Possible Values	In log or derived	Sensor data or derived
my_position.x	double	-1.37 to 1.37	In log	Available via sensors
my_position.y	double	-0.76 to 0.76	In log	Available via sensors
my_side	enum	EAST or WEST	In log	Available via sensors
ball_distance	double	0 to 2.7	Derived from logged my_position and ball position	Available via sensors
angle_to_ball	double	$-\pi$ to π	Derived from logged my_position and ball position	Available via sensors
team_context	enum	EAST or WEST	In log	Available in team context
my_context	enum	APPROACH, KICK, WAIT	Derived from log data	Available in team context

Table 4-2 Enumeration of Actions

Action Name	Parameters
Move	Angle, speed
Stop	None
Kick	Angle
Transition to Approach	None
Transition to Kick	None
Transition to Wait	None
Transition Team Context to East	None
Transition Team Context to West	None

The team context will know to transition from one context to another when one of the robots transitions to awaiting turn. The player and ball positions are reported in two-dimensional (x,y) coordinates. The coordinates refer to a position on the soccer field. The TeamBots coordinate system defines the middle of the field as 0,0. The x-coordinate can have a value between -1.37 to +1.37. The y-coordinate can have a value from -0.76 to + 0.76. The players are differentiated by the side of the field they inhabit. The possible values are East and West. The actions supported by TeamBots for each robot are the ability to set a steer heading and speed. The possible values for the steer heading are angular values from 0 to 2π where 0 is due East and π is due West. The possible speed values are 0, 0.5 and 1.0. The robot can also choose to kick. This kick will always be in the currently-faced direction.

Using this information and knowledge about the game, it is possible to for the developer to manually derive the contexts of the robots. Two team contexts are defined: East turn and West turn. This indicates which robot's turn it is to kick the ball. These data are logged from a coordinating class in the original game. These log data can be used to determine team context. At run-time, a change in team context is triggered by a robot kicking the ball. The developer can analyze the individual behavior by watching the game and looking at the training data. This process led to the individual contexts shown in Figure 4-3. In this figure, the contexts derived are: APPROACH_BALL, KICK_BALL, and AWAITING_TURN. In order to determine context, the first piece of data from the log that is analyzed is the information on whose turn it is. If this is the East robot, and it is East's turn, then AWAITING_TURN is eliminated as a possible context. Since kicking the ball is a useless endeavor when the ball is not within kicking distance, the differentiation between the remaining contexts is made by the proximity of the robot to the ball. If the ball is within kicking distance and the robot is facing toward opposite side of field, the context becomes KICK_BALL. Otherwise, the context is APPROACH_BALL. Table 4-1 shows the representation of the data elements contained in the situation vector. The table contains information on data element names, possible values, data type and how determined during training and run-time. Some elements can be obtained directly from the log file and sensors, other elements are derived by simple mathematical calculations or decision trees. A Java implementation of this data structure is shown in Figure 4-4.

```
public class Situation
{
      public enum SIDES {EAST, WEST};
      public enum IND_CONTEXT { APPROACH, KICK, AWAIT_TURN};
      public double myPositionX;
      public double myPositionY;
      public SIDES my_side;
      public double ball_distance;
      public double angle_to_ball;
      public SIDES team_context;
      public IND_CONTEXT my_context;


}
```

Figure 4-4 Java Implementation of Situation Vector

4.6.2 Learning Module

The next step in COLTS is to create the behavior maps for each context. This does involve writing code unique for each domain. The main Java class of this application is called Trainer. The Trainer class must be modified to read the text logs output from the TeamBots simulation and translate it into Situation classes containing the elements described in Table 4-1 for each time-step. Once a Situation vector is built for a log entry, the actions taken by the robot following that situation are determined by examining the log further. Each time a Situation vector and action vector are paired, the pairing is stored into the behavior map for the context defined in the Situation vector. Once all log entries have been parsed, each behavior map is saved to a file named for the context.

4.6.3 Run-time Module

The run-time classes associated with COLTS are largely reusable. The reusable classes are defined in a UML class diagram in Figure 4-5. Before execution, the Agent class must be modified to add the application-specific code to get the sensor inputs of each team member and build a current Situation class. The Team Context must also be modified to instantiate an instance of each of the defined individual contexts. The Team Context also needs a file to accept the name of an individual context from an agent and return an instance of that context.

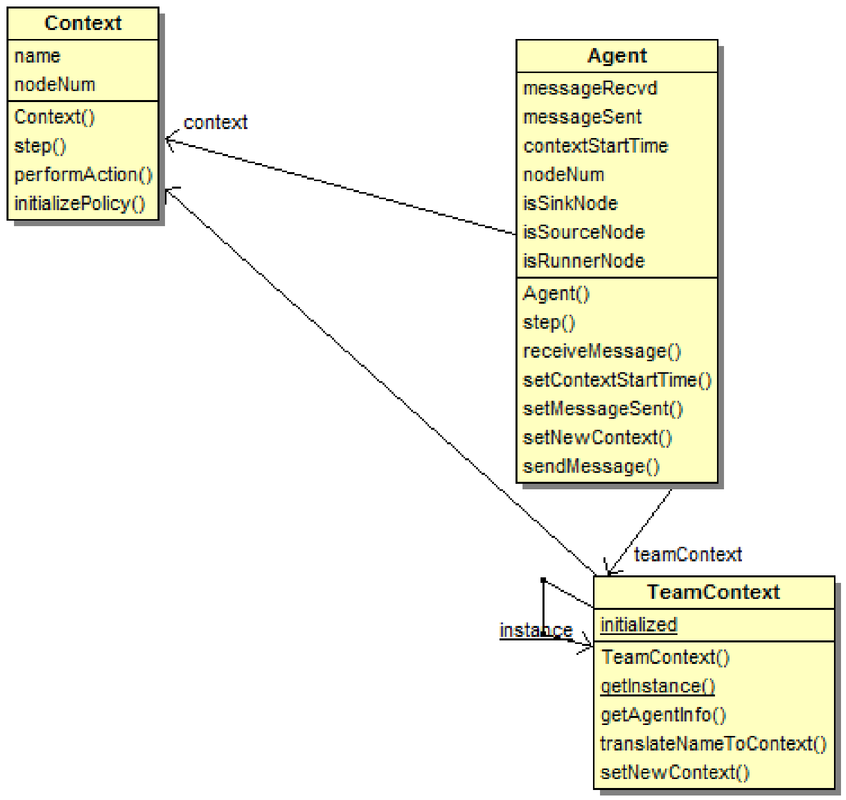


Figure 4-5 Reusable COLTS Run-time Java classes

At run-time, the simulation calls the Agent class step() method. The Agent class pulls in sensor data from the simulation and context data from the Team Context class to build a Situation vector for each time-step. Once the vector is built, the appropriate context class is called. Within the context, the current situation is compared to existing situation in the behavior map and the closest one found. The contextually-weighted nearest-neighbor algorithm must be coded to find the individual differences for all the data elements in the Situation Vector, S. The contextual weights may need some adjustment at this point. This should be done initially using some common domain knowledge. For example, when a player is in context AWAITING_TURN, the ball distance and angle weights are irrelevant to processing and can be set to 0. However, team context weight is quite important because the individual context will change once the team context indicates that this robot is now authorized to kick the ball. In more complex simulations, it may be necessary to adjust weights further using trial and error. Upon completion of weight adjustment, the newly-trained COLTS agents are ready to play “catch”.

4.7 Experiment Plan

COLTS provides a generic approach to observing teamwork behavior and creating a multi-agent software system capable of replicating the observed behavior. In order to demonstrate the generic nature of the approach, three prototypes were developed using COLTS. One of the drawbacks of the memory-based behavior policy used is that the more complicated the problem, the larger and more elaborate the state and action vectors become. Therefore, it makes sense to begin with a minimally-complex application upon which to experiment and work up to more complex tasks. A minimally-complex application will have a small situation vector and the number of possible situations and actions will also be finite and relatively small. Another issue is the difficulty in acquiring examples of effective teamwork that have adequate information associated with them. In addition, the use of human teams with limited availability also hampers the ability to get enough original samples to make a true statistical analysis. For that reason, this research uses simulated teams as examples to develop prototypes of COLTS. The simulated teams chosen have strengths and weaknesses just as an actual human team would, but with the additional ability to produce as much training and evaluation data as is needed so that significant tests of transferred behavior can be made.

The first team chosen to observe and prototype is a simulated bucket brigade. There are only four types of team members for the chosen simulation and the number of behaviors is minimal. This offers the ability to develop, implement and debug the run-time frameworks, behavior maps and training algorithm without great complexity. The behaviors to be observed and learned are so simple that if the COLTS approach were to not work with this type of teamwork, it would have quickly shown the COLTS approach to be fatally flawed.

The second prototype involved learning from a grid-based pursuit-evasion game. The team being observed was a four member pursuit team attempting to capture a single evader. A 30x30 grid offered significantly more potential positions and situations than the one bucket brigade prototype, which is one-dimensional. Although the number of teammates is not significantly larger, the communication amongst the team members is more complex. The size of the action vector is also increased. In the bucket brigade, communication is limited only to the team members on either side of a particular player. In the pursuit-evasion game, the effectiveness of the team depends upon insight into the positions and goals of all teammates and the sharing of insight into the opponents' position. These additional complexities gave insight into the effectiveness of the learning algorithm and provided an opportunity to fine-tune the training and nearest-neighbor algorithms.

The third prototype developed learned from a simulated soccer team in the TeamBots [Balch, et al, 2000] simulation. TeamBots was developed to test software for robots entering the RoboCup small-size soccer league competition. The five member teams and simulated soccer pitch provide an excellent testbed for stressing the algorithms. The larger number of teammates and opponents increases the complexity of the situation vector and the variety of actions available to perform is much greater than in either of the first two prototypes. In the small-size league, each robot agent has insight into all of the players on the field. This requires each agent to track position and heading information for themselves, the ball, four teammates and five opponents. This prototype also requires that the behavior function return actions within a predetermined time in order for the game to be played in real-time. This means that the nearest-neighbor distance calculations for all five teammates must occur within approximately 20 ms or the game will be held up. In order to achieve the time goal, the size of the behavior maps must be kept as small as possible and nearest neighbor calculations be kept to a minimum as well. This is a balancing act, as a too small behavior map will be unable to generalize well. Therefore, it is necessary to manually tune the size of the behavior maps by adjusting the amount of training data input into the learning algorithm.

Each prototype was evaluated for effective transfer of skills. First, the prototype is tested under the same conditions as the observed team. If COLTS is effective, the prototype should be able to perform the same task as efficiently as the observed team. This requires that the task performed be measurable in some way. A typical measurement can be the achievement of a goal or set of goals within a set amount of time. The prototype was also evaluated for the generalization of the transferred skills. This places the prototype in a new, untrained scenario and again measures its achievement against the observed team. The same measurement used for the training scenario can be used.

The observed teams from which teamwork and individual skills are learned are discussed in detail in Chapters 5 through 7, one for each of the prototypes. The application-specific implementations of the CCxBR framework, training algorithm and contextually-weighted nearest neighbor algorithm are presented there as well. The testing types and results of each individual prototype and conclusions drawn are also described in these chapters. Chapter 8 presents overall conclusions from the experimentation and possible future work based on this approach to learning teamwork by observation.

CHAPTER 5 PROTOTYPE 1: BUCKET BRIGADE

The bucket brigade is an example of teamwork with a small situation vector, a limited number of possible situations and limited number of actions. This example of a production team is the basis of the modern assembly line. A bucket brigade is a method of transporting buckets of water from a low-pressure water source to a location where it is needed, such as a fire, when a source of pressurized water is unavailable. A line of people is formed from the source to the fire. The person at the water source fills the bucket with water and passes it to the next person in line. The bucket is then passed from person to person down the line until the last person in line empties the bucket onto the fire. The emptied buckets can be transported back to the water source in one of two ways. A designated runner or runners can take the buckets from the end of the line back to the beginning. Another method is that after emptying the bucket, the person can then run back and take position at the beginning of the line and everyone advances a place in line. For purposes of this experiment, a brigade of varying sizes is simulated with a single runner to return buckets from the end to the beginning of the line. A log was created indicating current state and action taken for each member of the brigade. In this simulation, the brigade members were stationary with a runner moving the empty buckets back to the beginning of the line. The time to move a fixed number of buckets is used as the metric of team effectiveness. A team of COLTS agents were created from observation of the simulation. The COLTS team is considered effective if it can meet or exceed the count of buckets emptied by the observed simulated team. Generalization can be tested by increasing the number of brigade members and testing against a similar increase in the observed team.

5.1 Observer Module

The following subsections describe the processes and tools used in the observer module to create the new COLTS bucket brigade team agents.

5.1.1 Observed Team and Simulation Environment

The bucket brigade simulation chosen consists of a game portion that tracked status and executed the simulated actions, and the agents. Figure 5-1 shows a block diagram of the observed bucket brigade simulation. It shows that the simulation consists of a simulation engine portion that drives a simple GUI and communicates with agents representing the team members. The diagram also depicts the four types of agents developed for the simulation: source, sink, runner, and node agents. These were the portion of the simulation replaced with the prototype CCxBR team.

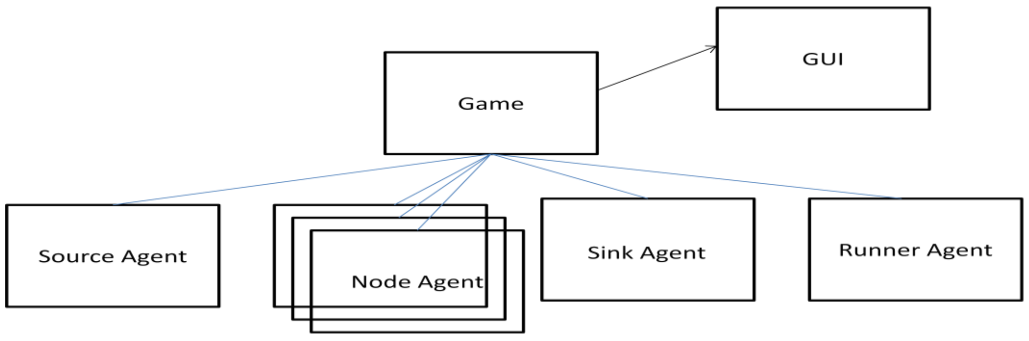


Figure 5-1 Observed Bucket Brigade Block Diagram

5.1.2 Training Data

The simulation in which the agents performed their task was instrumented to log data to a text-based log file. This instrumentation served as the observer module for COLTS and the log file served as input into the COLTS learning algorithm. The simulation output the state of each agent at the end of each time-step to the log.

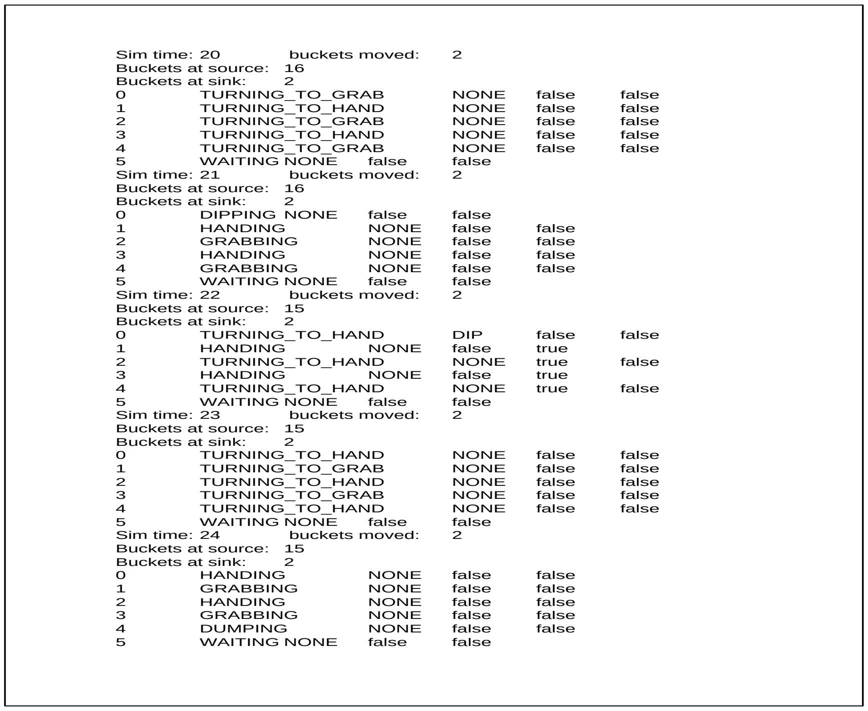
Table 5-1 Logged Data Types for each Time-step

Data Element	**Type**
Simulation time	Integer Seconds
Number of buckets at source	Integer
Number of buckets at sink	Integer
Brigade Member 1 State	State Data
Brigade Member 2 State	State Data
Brigade Member 3 State	State Data
Brigade Member 4 State	State Data
Brigade Member 5 State	State Data
Brigade Member 6 State	State Data

Table 5-2 Brigade Member State Data Types

Data Element	**Type**
Current Activity	DIPPING, DUMPING, HANDING, GRABBING, TURNING_TO_HAND, TURNING_TO_GRAB, WAITING, RETURNING_TO_SINK, RETURNING
Message Sent	True or False
Message Received	True or False
Action Taken	DIP, DUMP

The types of data logged each time-step are described in Table 5-1. Table 5-2 further defines the state data logged for each brigade member. A bucket brigade with six members was observed for the time it took to move 50 buckets of water from the source to the fire. An example of the log file output is shown in Figure 5-2. This figure represents the log entries from simulation time=20 to simulation time=24. Each entry is started by a line showing the simulation time and how many buckets of water have been dumped at the fire. These values are tab-delimited. The second line shows the number of empty buckets currently at the source and the third line indicates how many empty buckets are at the sink (the fire). The lines following are the current states of the various agents in the system.



```
Sim time: 20        buckets moved:      2
Buckets at source:  16
Buckets at sink:    2
0          TURNING_TO_GRAB          NONE     false      false
1          TURNING_TO_HAND          NONE     false      false
2          TURNING_TO_GRAB          NONE     false      false
3          TURNING_TO_HAND          NONE     false      false
4          TURNING_TO_GRAB          NONE     false      false
5          WAITING NONE     false      false
Sim time: 21        buckets moved:      2
Buckets at source:  16
Buckets at sink:    2
0          DIPPING NONE     false      false
1          HANDING          NONE     false      false
2          GRABBING         NONE     false      false
3          HANDING          NONE     false      false
4          GRABBING         NONE     false      false
5          WAITING NONE     false      false
Sim time: 22        buckets moved:      2
Buckets at source:  15
Buckets at sink:    2
0          TURNING_TO_HAND          DIP      false      false
1          HANDING          NONE     false      true
2          TURNING_TO_HAND          NONE     true       false
3          HANDING          NONE     false      true
4          TURNING_TO_HAND          NONE     true       false
5          WAITING NONE     false      false
Sim time: 23        buckets moved:      2
Buckets at source:  15
Buckets at sink:    2
0          TURNING_TO_HAND          NONE     false      false
1          TURNING_TO_GRAB          NONE     false      false
2          TURNING_TO_HAND          NONE     false      false
3          TURNING_TO_GRAB          NONE     false      false
4          TURNING_TO_HAND          NONE     false      false
5          WAITING NONE     false      false
Sim time: 24        buckets moved:      2
Buckets at source:  15
Buckets at sink:    2
0          HANDING          NONE     false      false
1          GRABBING         NONE     false      false
2          HANDING          NONE     false      false
3          GRABBING         NONE     false      false
4          DUMPING          NONE     false      false
5          WAITING NONE     false      false
```

Figure 5-2 Bucket Brigade Data Log

In this case, there are six agents. They are identified by their node number. Each line contains five pieces of information that are tab-delimited. The first piece of information is the identifying node number, the second indicates the state of the agent at the end of the time-step, the third indicates any actions taken in that time-step, the fourth indicates whether a message was received from an adjacent agent and finally, the fifth piece of information indicates whether a message was sent. The term message is used to represent the interaction between the brigade members when a bucket is passed. A message sent indicates that a brigade member has taken a bucket offered by an adjacent member. A message received acknowledges that the offered bucket has been taken. The log file is complete and contains adequate information to construct a new set of COLTS agents driven by behavior maps.

5.1.3 Situation and Action Vector specification

As discussed in Chapter 4, the next step in the development of COLTS agents after obtaining a textual training log of data is the development of the situation vector S as a Java data class. The class must consist of data that are available to the brigade agent at run-time, and also available through the log file. Table 5-3 describes the data elements of the situation vector S. All of the data listed can be derived from the log file and at run-time, all the data except non-neighboring agent states is available. The column entitled source of data indicates whether it comes directly from log or is derived.

Table 5-3 Bucket Brigade Situation Vector Data Elements

Element Name	Description	Data Type
timeInContext	Amount of time the agent has been in this particular context.	integer
messageSent	Whether or not agent has sent a message to neighboring agent.	boolean
messageRecvd	Whether or not a neighboring agent has a sent a message to this agent.	boolean
bucketsAtSource	Number of buckets at water source.	integer
bucketsAtSink	Number of buckets at sink (fire).	integer
isSinkNode	Is this agent the final member before sink (fire).	boolean
isSourceNode	Is this agent the member closest to water source.	boolean
isRunnerNode	Is this agent the runner.	boolean
myState	Enumeration describing the context.	Contexts enumeration
nodeNum	Number assigned to brigade member (agent) by game.	Integer
agentStates[]	An array of all brigade members and their current context.	Contexts enumeration

Table 5-4 shows the enumerated types used in the situation class. The table also shows the enumeration of possible actions in the simulation. This enumeration includes actions within the simulation and context transfers as well. The context transition actions are driven by the contextual analysis described in the next section. The remaining actions are determined by the simulation. These are the available actions for each agent in the simulation. Each action and context transfer can execute within one time-step. The final enumeration is the message type sent between brigade members. In this simulation, there is only one type of message sent and received.

Table 5-4 Enumerations

Enumeration	Values	Description
Contexts	GRABBING, DIPPING, TURNING_TO_HAND, HANDING, TURNING_TO_GRAB, DUMPING, WAITING, RETURNING, DUMPING_BUCKETS, RETURNING_TO_SINK, UNKNOWN	The names of the various contexts in the system.
Actions	NONE, DUMP, DIP, TAKE_BUCKET, TAKE_BUCKETS, RETURN_BUCKETS, TRANS_GRABBING, TRANS_DIPPING, TRANS_TURN_HAND, TRANS_HANDING, TRANS_TURN_GRAB, TRANS_DUMPING, TRANS_WAITING, TRANS_RETURNING, TRANS_BUCKET_DUMP, TRANS_RETURN_SINK	Possible actions for an agent to take within a time-step. Enumerations beginning with TRANS indicate that an action is a context switch.
Message Types	TAKE_BUCKET	Coordination message

Each data element of the Situation Vector described in Table 5-3 can be mapped directly to a data element in the data log. This mapping is shown in Table 5-5. The data elements can also be obtained at run-time by method calls to the simulation or from agent state data. The actual situation vector is implemented as a Java class with public data elements of the types designated in Table 5-3. To represent the action vector, an enumerated data type containing the enumerations described as Action in Table 5-4 was used.

Table 5-5 Mapping of Log Data to Situation Vector

Situation Vector Name	Logged Data Name	Derivation Required?
timeInContext	Simulation time	No
messageSent	Brigade Member State	Yes, match node number to Brigade Member number
messageRecvd	Brigade Member State	Yes, match node number to Brigade Member number
bucketsAtSource	Number of buckets at source	No
bucketsAtSink	Number of buckets at sink	No
isSinkNode	Brigade Member State	Yes. Does node number match Sink Agent node number
isSourceNode	Brigade Member State	Yes. Does node number match Source Agent node number
isRunnerNode	Brigade Member State	Yes. Does node number match Runner Agent node number
myState	Brigade Member State	Yes. Match node number to Brigade Member number
nodeNum	Brigade Member Number	No
agentStates[]	Brigade Member 1-6 States	Yes, only neighboring states used in situation vector

5.1.4 Contextualization

The next step of developing the learning algorithm for the bucket brigade was to manually analyze the observed simulation and determine the appropriate contexts for the COLTS agents. While examining the logged data, it became clear that a natural contextualization existed in the agent's current activity. Each activity represented a unique context of the agent. The various contexts of the observed agents are shown in Figure 5-3, 5-4, 5-5, and 5-6. These figures show a contextual flow for each agent in the observed simulation. This analysis is used to determine the various contexts in this particular application. As a result, a total of 10 contexts were determined to exist in the simulation. These contexts are enumerated in Table 5-4.

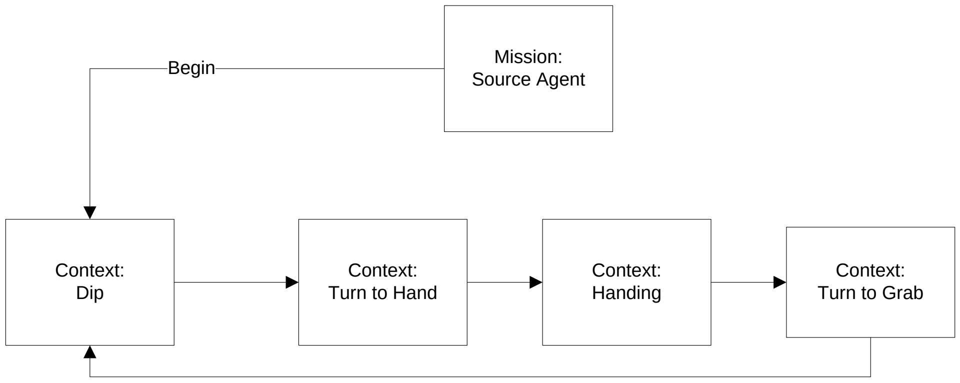


Figure 5-3 Context Analysis Source Agent

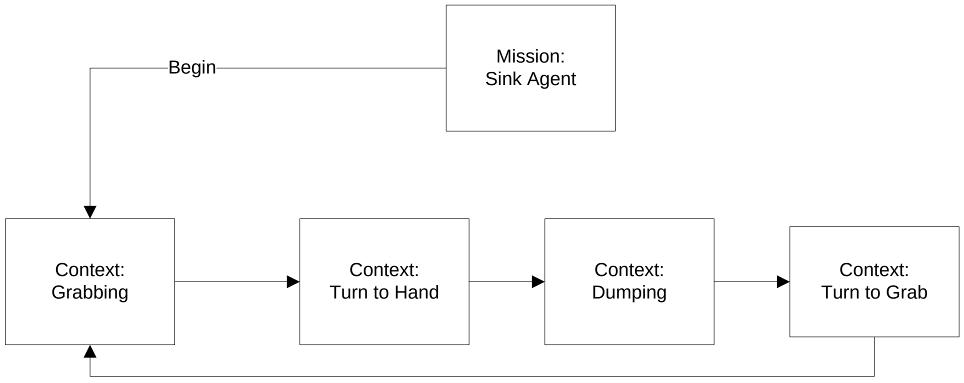


Figure 5-4 Context Analysis of Sink Agent

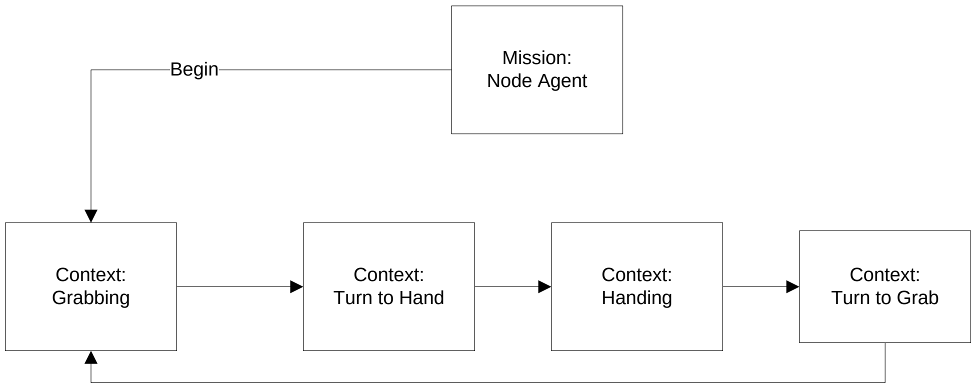


Figure 5-5 Context Analysis of Node Agent

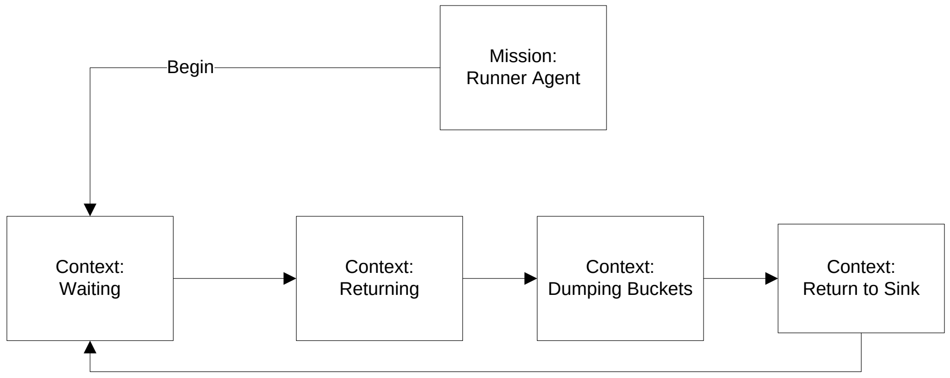


Figure 5-6 Context Analysis of Runner Agent

These contexts were determined by manually examining the parsed states of each agent in the log file. The actions defined in the observed simulation easily translated into actions for the prototype. Actions indicating a transfer to a new context were added. The enumerations of the actions, states and messages are defined in Table 5-4. Only one team context was needed. The team goal is always to move as many buckets of water from the source to the sink as quickly as possible. The determination and designation of the contexts along with the situation and action vectors provide the input needed to begin the learning portion of the COLTS system.

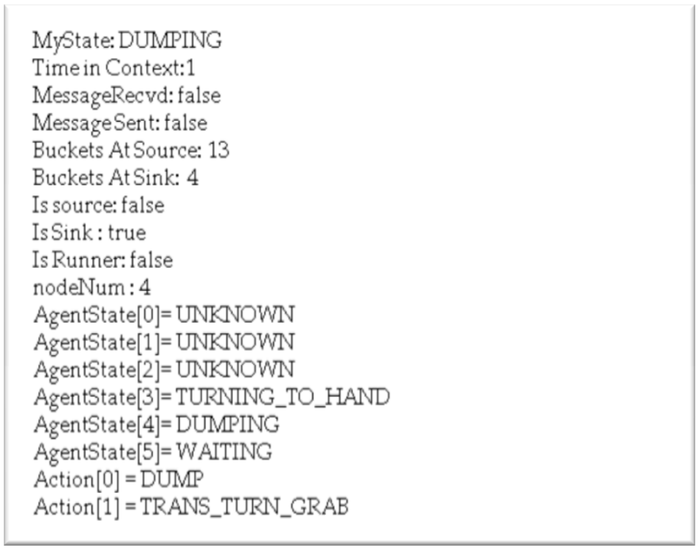
5.2 Learning Module

The learning module creates the behavior maps for the COLTS bucket brigade members. It was implemented as a stand-alone Java application. Two Java classes are shared between the learning algorithm and run-time agents. These are a Constants interface class and a Situation class. The Constants interface contains the enumerations of the contexts and actions as described in Table 5-4. The Situation class is an implementation of the situation vector S. The complete source code for the Constants and Situation Java classes is available in Appendix A. Once these classes were created, the log file parsing was implemented in the main class of the learning module named Trainer. Like the Situation and Constants class, the code for this parsing is specific to the log file output by the observed bucket brigade simulation and is not reusable for other applications of COLTS. The source code for the parsing is also available in Appendix A.

The learning algorithm was implemented in a Java class called Trainer. The algorithm shown in chapter 2 is used as the basis of this class. The parsing of the log file from text to data elements in the Situation class is one of the elements that is specific to this prototype. The second is the determination of context from the data parsed in the log. For this application, that determination was not difficult to implement. For each time-step, a textual description of the current state is given. These are translated directly into one of the contexts listed in Table 5-4. A method of the parsing algorithm is designed to translate from the textual state description into an enumerated context value.

Figure 5-8 shows a high-level flow chart of how the learning algorithm was implemented for the bucket brigade. The algorithm handles each brigade member in turn. For each brigade member, the log file is reset to the beginning and iterates through each time-step logged. As the data for each time-step is read in, the parsing code creates an instance of the Situation class for the brigade member currently being examined. Next, the context of that Situation instance is determined from the state description in the log file. The log file also contains certain actions that occurred during that time-step. The actions that are can be parsed from the log and directly associated with a Situation are: DUMP, DIP, TAKE_BUCKET, TAKE_BUCKETS, and RETURN_BUCKETS.

Determining a context transfer action is more complicated than simply parsing the log file. Because each time-step represents the state of the agent at the end of the time-step, the Situation from the prior time-step is used to determine the context of the agent at the beginning of the time-step. The Situation instance at time t is compared with the Situation instance at time t-1 to determine whether or not a context switch occurred. A context switch has occurred if the agent context at time t is different than the context at time t-1. If a context switch did occur, the appropriate action enumeration is added to the action list associated with time t-1. For each Situation class parsed, there could be from zero to two actions associated with it. The first possible action would be one of the actions parsed from the log file. The second possible action would be a context transition action. Once, the Situation class was paired with a set of actions, it was stored in the behavior map for that context. Behavior maps were implemented as Java Hashmaps with the key being an instance of the Situation class and the data being the set of actions.



```
MyState: DUMPING
Time in Context:1
MessageRecvd: false
MessageSent: false
Buckets At Source: 13
Buckets At Sink: 4
Is source: false
Is Sink : true
Is Runner: false
nodeNum : 4
AgentState[0]= UNKNOWN
AgentState[1]= UNKNOWN
AgentState[2]= UNKNOWN
AgentState[3]= TURNING_TO_HAND
AgentState[4]= DUMPING
AgentState[5]= WAITING
Action[0] = DUMP
Action[1] = TRANS_TURN_GRAB
```

Figure 5-7 Textual Representation of Behavior Map

For the bucket brigade, the majority of the actions taken were context transfers as they moved from one activity to another. It was not necessary to utilize an algorithm to resolve conflicting actions for the same situation as the bucket brigade members consistently performed the same actions given the same situation. Once the Trainer class completes the process of parsing for every agent at every time-step, the behavior maps are saved to individual files.

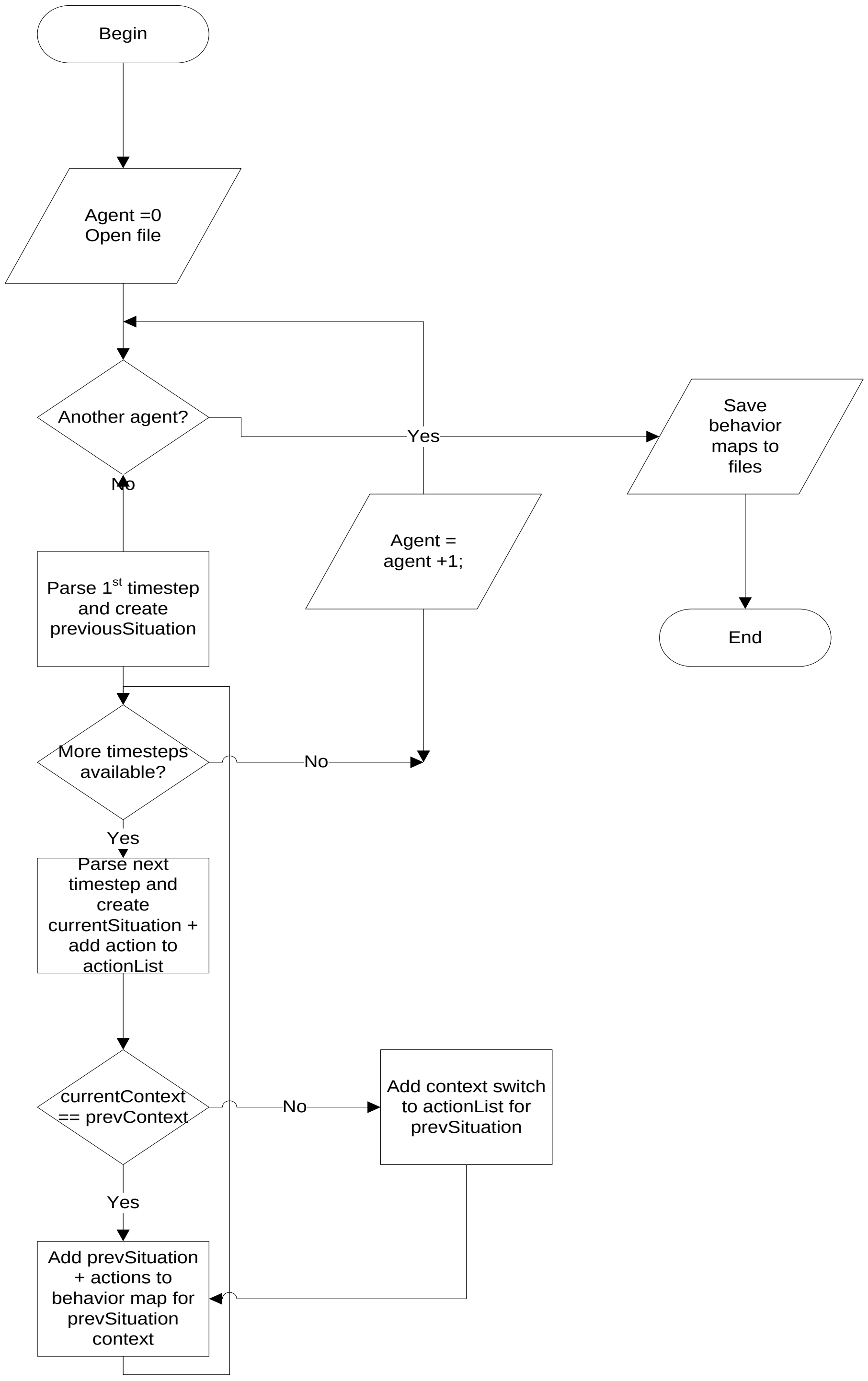
The files are named for the context they represented. For example, the behavior maps for the "HANDING" context is named "handing.map". While the map files themselves are saved as binary files, debugging code was added that also outputs a textual representation of the behavior map. Figure 5-7 shows an entry from the textual representation of the DUMPING context behavior map. The majority of the entry is data elements of the Situation class. The last two entries are the two actions to be taken when this situation is chosen.

5.3 Run-time Module

The run-time module consists of the agent implementation and the contextually-weighted k-nearest neighbor implementation. These are described in the following sections.

5.3.1 Agent Implementation

The next step is to create the COLTS agents to replace the preprogrammed agents in the observed simulation. The COLTS agents created to learn from the preprogrammed agents were designed to appear programmatically identical to the observed agents from the viewpoint of the game engine. Figure 5-9 shows the new prototype bucket brigade block diagram. It was not necessary to alter the simulation engine and GUI to work with the new COLTS agents. However, a COLTS framework was implemented to replace the observed hard-coded agents of the observed bucket brigade. The Java classes created for the framework were simply named: TeamContext, Context, and Agent. Each class is designed to be as reusable as possible between COLTS applications.



Begin
Agent =0
Open file
Another agent?
Yes
No
Save behavior maps to files
End
Agent = agent +1;
Parse 1st timestep and create previousSituation
More timesteps available?
No
Yes
Parse next timestep and create currentSituation + add action to actionList
currentContext == prevContext
No
Add context switch to actionList for prevSituation
Yes
Add prevSituation + actions to behavior map for prevSituation context

Figure 5-8 Bucket Brigade Trainer Flow Chart

A UML diagram in Figure 5-10 shows the relationship among these classes and the Situation and Constants classes that are common with the Trainer class. While not shown in this diagram, an instance of Agent is created for each team member. The behavior maps output by the learning algorithm are input into their associated Context instance when the class is instantiated via the InitializePolicy method of the class. The team context within the Run-Time framework was created to maintain the team's goal and group situational awareness. It is a singleton class, meaning that only one instance of the TeamContext exists in any given application. The TeamContext class instantiates all the Context class instances, maintains context information about all agents and contains a method that converts the action vectors obtained from the behavior maps into action.

The COLTS agents are all identical. At instantiation, each is given a unique node number and starting context. The starting context and node number provide the information necessary for the agent to determine its role on the team. Like the agents, the implementation of each context instance is identical. The difference between context objects is in the name and the behavior map. These maps are linked to a particular context and contain a mapping of situation to actions appropriate in that context. The behavior maps are the output of the training algorithm stored in binary files. The step() function of the class compares a passed-in instance of Situation and returns the action vector associated with the closest Situation in the behavior map.

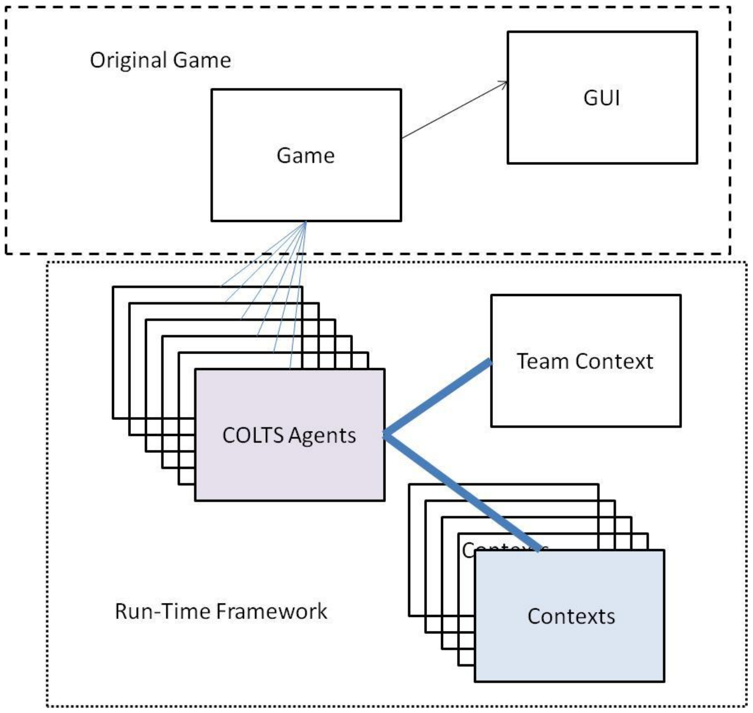


Figure 5-9 Bucket Brigade with CCxBR Framework

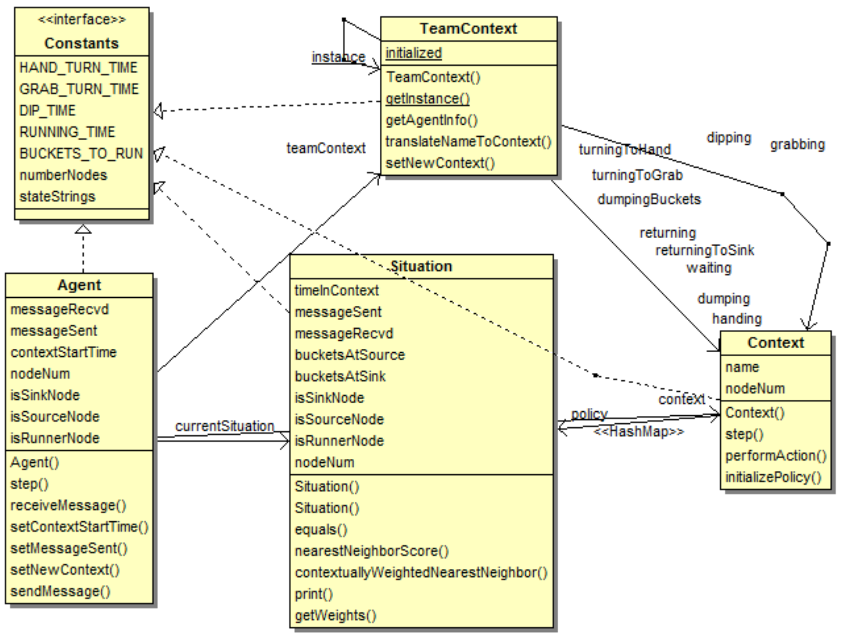


Figure 5-10 UML Diagram of CCxBR Framework

At runtime, the step() method of each agent is called by the game's SimEngine during each time-step just as the observed hard-coded agents were called. The code within this method then calls the step() method of the current context as shown in Figure 5-11. When the step() method in the agent is called, a data structure containing the current situation is built. That data structure is compared to the situations stored within the context object's behavior map for the closest match. When that match is found, a list of actions is returned. The actions are nothing but a list of enumerations that map to the behaviors supported by the game. The context implementation contains an inherited method capable of translating from the action enumeration to the appropriate game engine call or context transfer. This method must be hard-coded to the specifications of the unique domain.

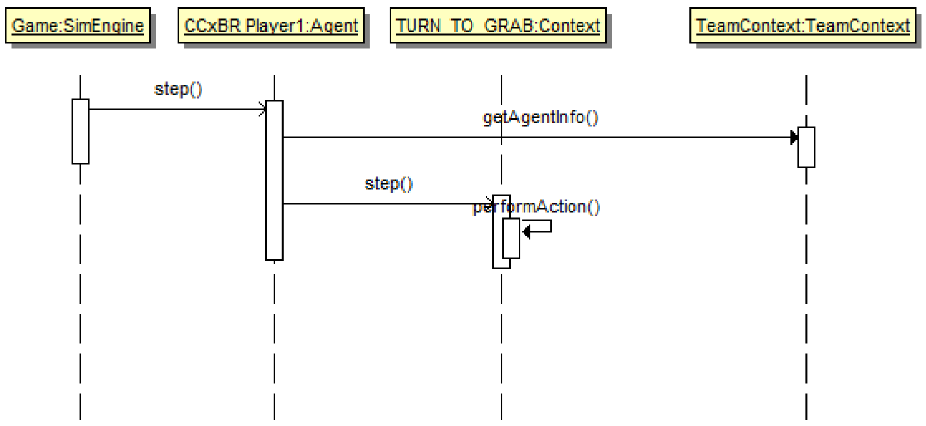


Figure 5-11 UML Sequence Diagram of Agent time-step

5.3.2 Contextually-weighted k-nearest-neighbor Algorithm

Finally, the run-time nearest neighbor algorithm and appropriate weight vectors were developed. For this application, the state vectors being compared were two Situation classes with a variety of data types. A member method of the Situation class provides the evaluation of the "closeness" of two situation classes using a nearest-neighbor algorithm. Initially, a single k-nearest-neighbor algorithm was developed for all contexts of the agents with a single weight vector. However, deriving the weight vector for the evaluation proved quite difficult. The set of weights that resulted in good behavior for the agents passing buckets was unable to duplicate the behavior of the agent running buckets from end point to beginning. To solve this problem, multiple sets of weights were determined based on the current context. This contextually-weighted nearest neighbor algorithm was successful in creating agents able to duplicate the behavior of the observed bucket brigade. The weight vectors were set manually using trial and error.

This trial and error method involved a little bit of common sense domain knowledge. The common sense portion came when looking at the behavior of each type of observed agent and analyzing which data were relevant to the current context. For example, the runner agent had no need to know the state of adjacent agents. This agent was strictly interested in how many buckets were currently sitting at the end of the brigade and whether or not there were enough to make a run back to the beginning worthwhile. Data elements in the situation vector relating to messages and adjacent agents could be given a weight of 0 in contexts used by the runner agent. Conversely, a bucket brigade agent involved in passing buckets has no need to care how many buckets are currently at the end of the line. Only the source agent has a need to know how many buckets are available at the source, so only the DIP context has a non-zero weight for that data element. For remaining elements, an initial weight of one was used and if an agent did not behave as expected in a particular context as determined by looking at the data log, the weight was adjusted up or down until better behavior resulted.

Run-time processing of a simulation time-step is the same regardless of the current context. When the step() function of a context is called, the behavior map for that context is loaded. The current Situation is compared to each entry in the behavior map using the contextually-weighted k-nearest neighbor algorithm. The lowest scoring entry is chosen. The actions mapped to that entry are executed in that time-step. The k-nearest-neighbor equation is modified slightly to avoid performing square root calculations and use a set of weights related to the current context. The modifications simplify the implementation and offer a small performance improvement without losing precision. The new equation is as follows:

$$d(x, q) = \sum w_{cj} * abs(x_j - q_j)$$

where x equals original Situation class, q equals the compared against Situation class and w_c is the weight vector for a given context. Each data member of the Situation class is an element in the associated state vector. There is no magic formula to develop the weight vectors. Knowledge of the actions that normally occur during each context provides insight into which data are important in each context. The final weight vectors are shown in Tables 5-6 and 5-7 Table 5-6 shows the weights that remain constant regardless of context. Table 5-7 .shows the context specific weights and the contexts that use them. These weights have a value of zero for all other contexts.

Table 5-6 Common Weight Vectors for COLTS Bucket Brigade Prototype

Weight Name	**Weight Value**
My state	300
Is runner node	100
Is source node	100
Is sink node	100

Table 5-7 Context Specific Weights

Weight Name	Weight Value	Context
Message recvd	300	HANDING
Left neighbor state	100	GRABBING
Right neighbor state	100	HANDING
Time in context	100	RETURNING, TURNING_TO_HAND, TURNING_TO_GRAB, RETURNING_TO_SINK
Buckets at source	1	DIPPING
Buckets at sink	10	WAITING

5.4 Experimental Results

By using the weights associated with the context of the agent, the nearest-neighbor algorithm was able to accurately choose the correct actions for the agent. The COLTS prototype bucket brigade was able to exactly match the efficiency of the observed brigade. Since the simulation engine of the observed bucket brigade was used, the prototype output a log file in the same format as the observed simulation. The output of the prototype was matched line for line with the original training data. The two files were identical, indicating that the prototype bucket brigade was able to perform identically to the observed given the same situation. Each was able to move 50 buckets in 307 simulation time-steps. Repetition showed that the behavior of the prototype to be consistent from run to run. The results of all the testing is summarized in Table 5-8.

Table 5-8 COLTS Bucket Brigade Testing Results

Number of Members	Observed Team Time	COLTS Team Time
5	307 seconds	307 seconds
6	309 seconds	309 seconds
5 (different start context)	308 seconds	308 seconds
5 (with random turn times)	346 seconds	308 seconds
20	337 seconds	336 seconds

Next, the prototype was tested to see whether the learned behavior could be generalized to different starting conditions. Without repeating the training process, the prototype was given an additional team member in the bucket brigade. The resulting simulation was able to efficiently perform the task given. The observed simulation was also given an additional team member and the time to move 50 buckets from beginning to end was used as the metric for determining efficiency. Both the observed and prototype bucket brigades were able to move the 50 buckets within 309 seconds. The slight increase in time is consistent with the extra time needed to pass the first bucket down the longer line of team members. This test showed that the prototype was able to generalize the behavior of the observed team to duplicate the behavior when additional team members were added. This test stressed the contextually-weighted nearest-neighbor algorithm because no example behavior for the new team member existed. The new team member had to generalize behavior from similar team member in order to perform effectively. Since any additional nodes would be identical to the single one added in this test, there is every reason to believe that any additional nodes added would also behave identically to the observed simulation.

A second generalization test involved changing the starting context of the agents moving buckets. Initially, the source agent was initialized as turning to grab a bucket for dipping and the remaining agents were poised to grab a bucket as soon as it was available. In this test, the source agent was initialized in a dipping position and the other members were initialized in a turning to grab position. There were not any other choices as the remaining contexts assumed that the agent was already holding a filled bucket. So these were the only other choices for a realistic starting state. Again, the performance of the prototype was matched against the observed brigade in the same configuration. This time it took 308 seconds to move the 50 buckets.

These very exact results reflect that fact that observed team is perfectly efficient. Each action taken always took the same amount of time and no motion was wasted. The prototype agents trained from this example showed the same efficiency. While this was an excellent test of the CCxBR framework's ability to replace the observed agents and use the trained behavior maps, it was not a true representation of actual human behavior that is rarely perfectly efficient. A team of real humans performing this task would probably have some variations in the time it took to move a bucket from place to place.

For this reason, the experiment was repeated using a variation of the observed team that incorporated some variability into the timing of the movements of each agent. The new observed team uses a random number generator to determine if an agent will take two or three seconds to turn. If the random number generator uses an even distribution, the agent should turn in two seconds half the time and three the other half.

The new bucket brigade with random behavior takes 346 seconds to complete a 50 bucket run. Subsequent runs showed variations on this time, but the observed run at 346 second was chosen to train the new prototype. The prototype trained with these data was able to complete it in 308 seconds indicating that the nearest-neighbor algorithm consistently chose the more efficient turning path. Repetitions of the prototype showed that this was consistent behavior and moving 50 buckets always took 308 seconds.

During the testing, some observations of execution time were made. Specifically, the time to run the behavior functions of the COLTS agents was measured at the various team sizes. The goal was to determine whether or not the execution time truly did scale no more than O(n) where n is team size. It would be expected that the bucket brigade would not be a worst-case scenario and scale linearly. This is due to the fact that the individual team members do not have access to information about all their teammates. Access is limited to the teammates on either side of a team member. Therefore, the size of the situation vector does not increase with the size of the team. The times given are worst case found in a short run of about 10 seconds since the times were not exact every cycle due to operating system usage. Measurements were made in milliseconds. The worst-case scenario is expected to be O(n), but the results showed that the actual time was significantly less. If the run time was O(n), the expected execution time for teams of 12 and 18 would be two and three times the execution time for the team of 6. Table 5-9 shows the actual times detected. As expected, they are below the anticipated worst-case scenario of O(n).

Table 5-9 Execution Time of Various Team Sizes

Team Size	**Largest Execution Time (ms)**
6	0.420
12	0.500
18	1.220

The success of the original test and generalization tests indicate that the training algorithm and CCxBR agent framework are suitable for learning teamwork by observation. The original tests proved that the combination of the behavior map and the contextually-weighted nearest-neighbor algorithm were effective in transferring the behavior of the observed team to the new prototype. As expected, the first prototype behaved identically to the observed team in the identical situation. The prototype was also able to behave identically in similar situations as well. While the second prototype did not behave identically, the team still performed the expected behavior in an efficient manner. Because the behavior of the bucket brigade is simple in the sense that there are limited possible situations and actions, this prototype does not prove the effectiveness of this method in capturing collaborative behavior. However, it did provide an excellent test bed for developing the COLTS framework of agents, behavior maps and the contextually-weighted nearest neighbor algorithm. A second and third prototype using more sophisticated teams will offer more opportunity to weigh the usefulness of this machine learning approach.

CHAPTER 6 PROTOTYPE 2: PURSUIT-EVASION GAME

The second prototype developed used a pursuit-evasion game as its basis. In this game, a 30x30 two-dimensional grid contains a single "Red" submarine (the evader) and four "Blue" surface ships (the pursuers). At each step of the game, entities both the Blue ships and Red submarines can choose to move up, down, left, right or stay in their current space. The game ends when the four Blue entities are able to surround the Red entity, thereby preventing its future movement, or the Red entity is able to reach any edge of the grid, and thus escape. The COLTS prototype team learns its behavior from the Blue pursuit team. The Red evader remains the same for both observed and COLTS pursuit teams. In order to be effective, the Blue pursuit team must work together effectively to search the entire grid and locate the evading agent. Once the evader is detected by one of the pursuit team, the collaborative behavior continues with communication of the evader's position and seeking to attain the boxing positions.

The pursuit team is a more complex team than the bucket brigade team. Since each team member is aware of all the other team members, the situation vector size increases to contain the additional collaborative data involved with this four-member team. The number of possible situations is also significantly larger than the bucket brigade. Each team member must track positions for itself, its team member, and the Red evader. This gives a total of five entities to track each of which has 900 possible positions. There are also a slightly larger number of actions that each team member can perform in a given time-step. The larger number of possible situations requires more training data than the bucket brigade in order to adequately imitate the behavior of the observed team.

6.1 Observer Module

The following subsections describe the processes and tools used in the observer modules to create the new COLTS pursuit team agents.

6.1.1 Observed Team and Simulation Environment

The pursuit-evasion game has been a popular choice for testing distributed intelligence algorithms [Singh, 1993], and more recently, genetic algorithms [Nitschke, 2003]. There are several approaches to the pursuit team available. The observed team using an approach known as "Control distributed among altruistic peers". [Singh, 1993, p. 724] The four team members are all peers working together to trap the Red evader. Each team member has awareness of its own position and can detect other entities within a five step area. The team members are capable of communicating their own position and any detected entities to the other team members. Each team member has an assigned quadrant to patrol and an assigned position to box in the evader once it has been detected. Their operating instructions are straight-forward.

- When the evader's position is unknown, patrol the assigned quadrant in a search pattern designed to take full advantage of the sensors.
- When the evader's position is known either through its own sensors or message transmission from a team mate, compute the fastest path towards the evader and attempt to block the assigned position.

In both the observed and COLTS games, the evader agent moves randomly in the four possible directions. Approximately 10% of the time, it will remain stationary. Because of the pseudo-random nature of the Java random number generator, there is a slight bias to the Move Up action. Figure 6-1 shows a screenshot of the observed simulation.

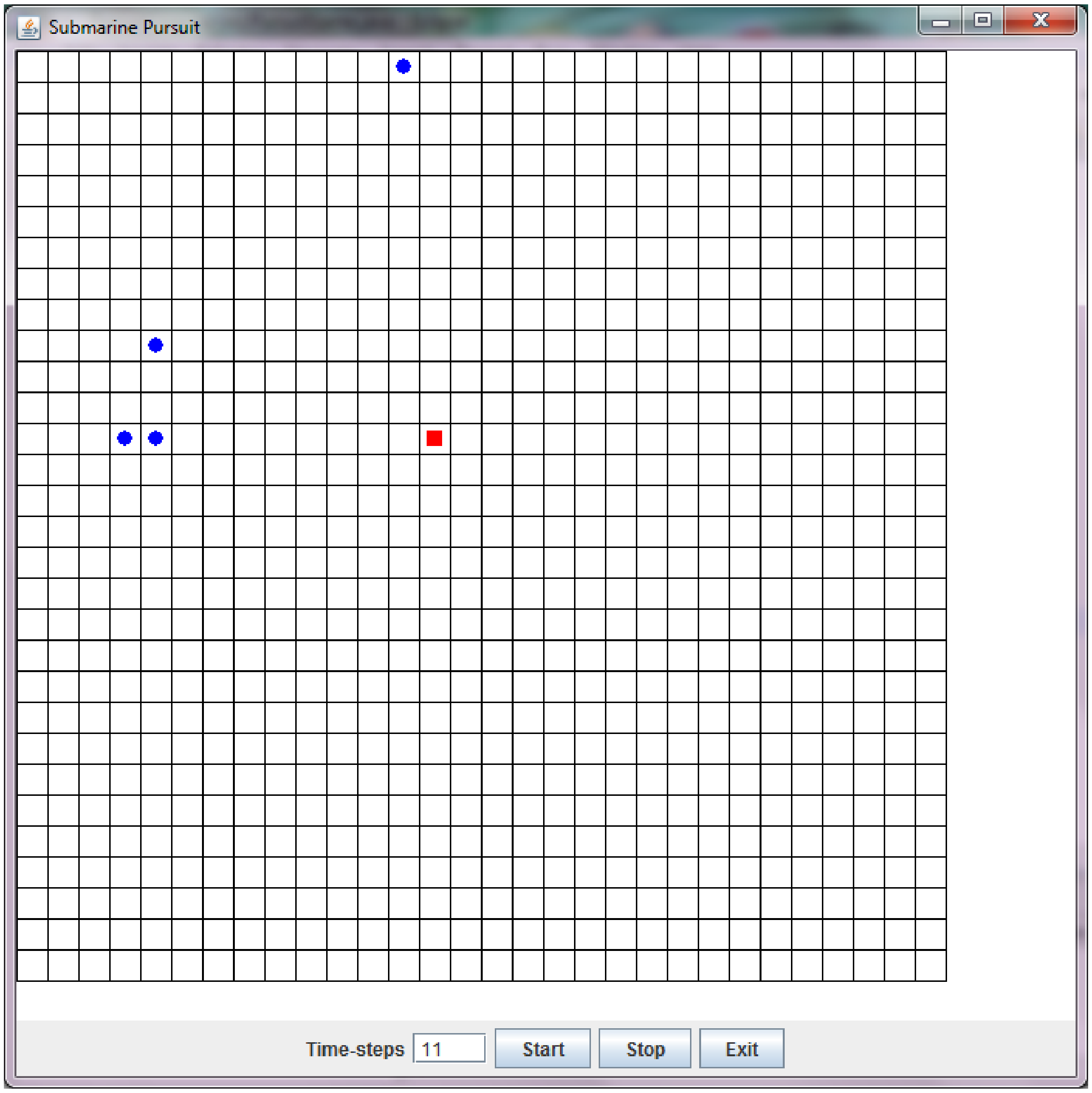


Figure 6-1 Pursuit-Evasion Game Screenshot

The Red evader is designated with a red dot in the grid and the four Blue pursuers are designated with blue dots in the grid. The basic infrastructure of the pursuit-evader game has some similarities to the bucket brigade as shown in Figure 6-2. The team member agents are separate classes and easily replaced by COLTS agents, just as in the bucket brigade game. The game is a discrete, time-stepped simulation and the Pursuit Game Engine calls each of the agents once each time-step and then updates the GUI to reflect the changes.

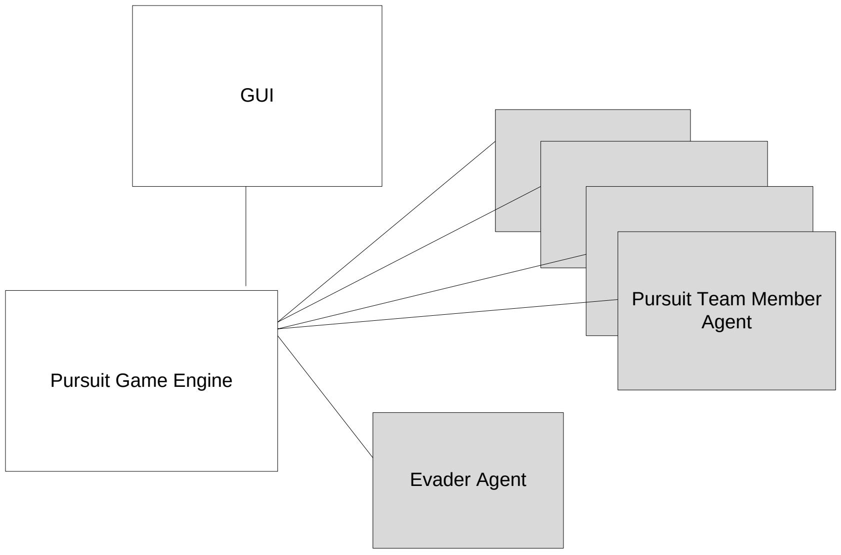
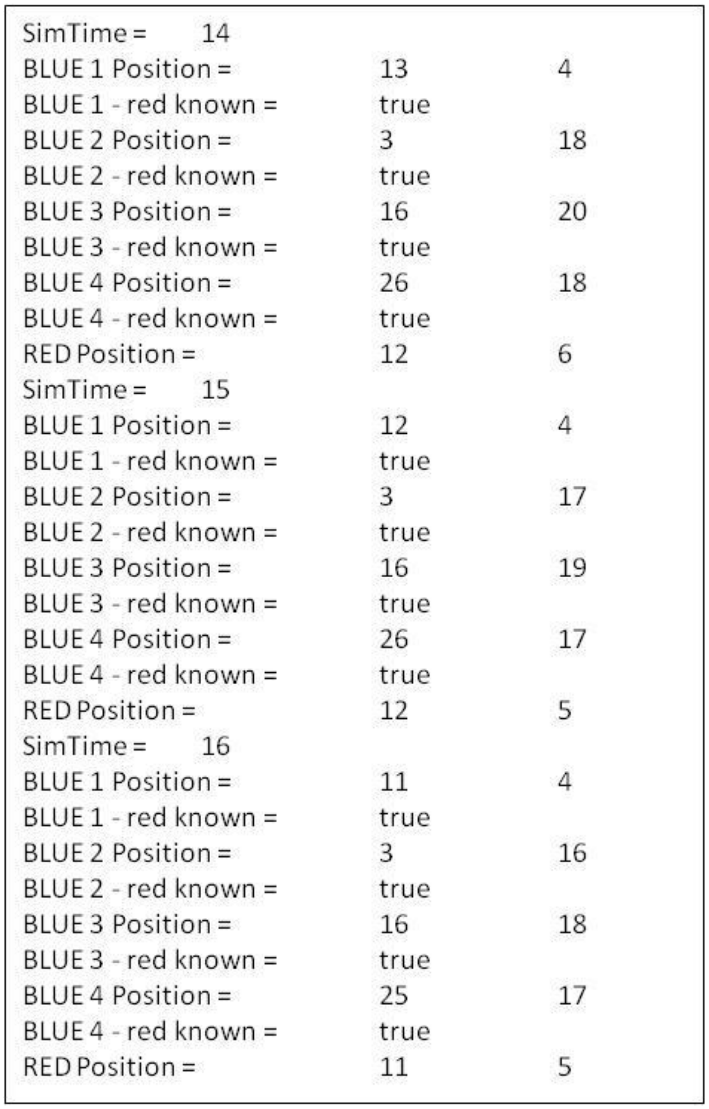


Figure 6-2 Observed Pursuit Game Block Diagram

6.1.2 Training Data

The observer module for COLTS was added to the Pursuit Game Engine and outputs data to a simulation log each time-step. The data logged at each time-step included simulation time, the evader's position, the pursuit team members' positions and whether or not the pursuers had detected the evading agent yet. An example of the data log showing three time-steps is shown in Figure 6-3. The data log entries are made at the end of each time-step, so the position shown is the result of any moves made within that time-step. Because the simulation is discrete, the position of the agents' does not move except during the time-step, so the moves made can be determined by comparing this position with the position at the last time-step.



```
SimTime =       14
BLUE 1 Position =               13              4
BLUE 1 - red known =            true
BLUE 2 Position =               3               18
BLUE 2 - red known =            true
BLUE 3 Position =               16              20
BLUE 3 - red known =            true
BLUE 4 Position =               26              18
BLUE 4 - red known =            true
RED Position =                  12              6
SimTime =       15
BLUE 1 Position =               12              4
BLUE 1 - red known =            true
BLUE 2 Position =               3               17
BLUE 2 - red known =            true
BLUE 3 Position =               16              19
BLUE 3 - red known =            true
BLUE 4 Position =               26              17
BLUE 4 - red known =            true
RED Position =                  12              5
SimTime =       16
BLUE 1 Position =               11              4
BLUE 1 - red known =            true
BLUE 2 Position =               3               16
BLUE 2 - red known =            true
BLUE 3 Position =               16              18
BLUE 3 - red known =            true
BLUE 4 Position =               25              17
BLUE 4 - red known =            true
RED Position =                  11              5
```

Figure 6-3 Example of Pursuit-Evasion Log

For the bucket brigade, a log file of one run through the simulation was used as input to the learning module of COLTS. This was sufficient for the bucket brigade with its limited number of possible states. The pursuit-evasion game has a much larger number of possible states and actions, so more data are necessary. The objective is to get the minimum amount of data needed to get good results. Since the simulation inserts 500 ms between time-steps to allow the GUI to be viewable by human eyes without excessive flicker, the size of the behavior maps and the amount of time to process them is of little concern in this simulation as long as the run-time agents combined take less than 500 ms. For that reason, fifteen different starting positions for the Blue team were chosen for use in collecting training data. The Red evader is always started in the center of the grid. The starting positions were chosen to make sure that each agent has a training data point over most of the grid. A total of ten games for each of the 15 starting positions were run and logged into a text file that was used as input to the learning algorithm. This creating a training file with 150 games total. The starting configurations were chosen from the thirty test cases listed in Table 6-1. The starting positions chosen for the training were: Test Cases 1, 2, 3, 4, 5, 6, 7, 8, 9, 13, 12, 18, 19, and 23.

Some test cases were specifically chosen to ensure that each blue ship starts in different quadrants, while others were chosen to enable comparison to previous implementations, and a few were simply chosen randomly. Table 6-2 shows the average percentage of blue wins for each test case using the random evader. In addition, the results are displayed for the same series of tests with the observed team against a more intelligent, *smart* evader. Since the COLTS prototype was trained only with games played against the random evader, these results against the smart evader can be used to test the prototype for generalization.

Table 6-1 Test Case Starting Positions

Test Case #	Blue1 Position (row, column)	Blue2 Position (row, column)	Blue3 Position (row, column)	Blue4 Position (row, column)	Red Position (row, column)
1	(0,0)	(0, 29)	(29, 0)	(29,29)	(14,14)
2	(0,0)	(0,1)	(0,2)	(0,3)	(14,14)
3	(0,26)	(0,27)	(0,28)	(0,29)	(14,14)
4	(29,0)	(29,1)	(29,2)	(29,3)	(14,14)
5	(29,26)	(29,27)	(29,28)	(29,29)	(14,14)
6	(0,28)	(0,1)	(0,2)	(0,29)	(14,14)
7	(0,28)	(29,28)	(29,29)	(0,29)	(14,14)
8	(0,0)	(0,1)	(29,0)	(29,1)	(14,14)
9	(0,28)	(0,29)	(29,28)	(29,29)	(14,14)
10	(0,0)	(0,1)	(29,28)	(29,29)	(14,14)
11	(29,0)	(0,0)	(29,29)	(0,29)	(14,14)
12	(0,29)	(29,29)	(0,0)	(29,0)	(14,14)
13	(29,29)	(29,0)	(0,29)	(0,0)	(14,14)
14	(0,14)	(7,14)	(21,14)	(29,14)	(14,14)
15	(14,0)	(14,7)	(14,21)	(14,29)	(14,14)
16	(0,0)	(7,7)	(21,21)	(29, 29)	(14,14)
17	(0,29)	(7,21)	(21,7)	(29, 0)	(14,14)
18	(5,8)	(21,25)	(27,29)	(2,4)	(14,14)
19	(18,18)	(7,11)	(2,3)	(9,9)	(14,14)
20	(7,20)	(3,9)	(23,26)	(29,7)	(14,14)
21	(0,29)	(1,28)	(2,27)	(3,26)	(14,14)
22	(1,5)	(1,10)	(1,15)	(1,20)	(14,14)
23	(16,16)	(17,17)	(18,18)	(19,19)	(14,14)
24	(13,13)	(12,12)	(11,11)	(10,10)	(14,14)
25	(29,3)	(5,19)	(16,18)	(2,2)	(14,14)
26	(29,11)	(0,11)	(0,16)	(29,16)	(14,14)
27	(4,28)	(28,4)	(26,7)	(7,26)	(14,14)
28	(3,15)	(7,15)	(19, 15)	(23,15)	(14,14)
29	(15,3)	(15,7)	(15,19)	(15,23)	(14,14)
30	(28,4)	(4,28)	(7,26)	(26,7)	(14,14)

Table 6-2 Observed Team Test Case Win Statistics

Test Case Number	Random Evader Blue Win %	Smart Evader Blue Win %
1	72.28	20.38
2	88.30	59.20
3	63.95	19.30
4	58.75	13.24
5	48.18	20.45
6	67.51	22.64
7	68.69	23.19
8	70.10	17.68
9	70.09	24.35
10	69.58	18.11
11	68.87	16.45
12	63.25	14.77
13	15.75	3.52
14	97.47	75.05
15	54.04	20.44
16	79.72	37.17
17	66.67	23.33
18	41.60	9.21
19	59.98	11.75
20	72.74	37.11
21	67.74	23.78
22	83.17	32.62
23	94.61	57.46
24	47.31	12.99
25	17.22	7.58
26	81.21	21.72
27	79.71	30.12
28	97.96	77.57
29	76.78	42.03
30	80.51	28.78

The smart evader moves randomly in the grid until a blue ship moves within detection range. Once it has been detected, the smart evader moves toward the nearest edge. In order to give the blue ships a chance to catch it, a small amount of random behavior is programmed into the evader even when it is moving toward the edge. Otherwise, the win ratio for the blue ships was often 0%, which did not allow for meaningful analysis of the trained agents. Because a test scenario with a set of randomly moving blue ships resulted in a win ratio of 0% against both the random and the smart evader, it is difficult to determine if behavior is truly learned with a very low win ratio.

Table 6-3 lists the data element logged for each time-step. Each time-step entry begins with a line describing the current simulation time. Following that are the entries for each Blue team member. Each Blue team member's position is given in x,y grid coordinates. Additionally, an indication of whether the team member is currently aware of the Red evader's position is given. The last line of each entry indicates the Red evader's position. This information is given whether or not the Blue team has detected the evader or not.

Table 6-3 Logged Data Types for each Time-Step

Data Element	**Type**
Simulation Time	Integer time-steps
Blue 1 Position X Coordinate	Integer
Blue 1 Position Y coordinate	Integer
Blue 1 Detected Red	Boolean
Blue 2 Position X Coordinate	Integer
Blue 2 Position Y coordinate	Integer
Blue 2 Detected Red	Boolean
Blue 3 Position X Coordinate	Integer
Blue 3 Position Y coordinate	Integer
Blue 3 Detected Red	Boolean
Blue4 Position X Coordinate	Integer
Blue 4 Position Y coordinate	Integer
Blue 4 Detected Red	Boolean
Red Position X Coordinate	Integer
Red Position Y Coordinate	Integer

This data matches the information available to the Blue team at run-time. They have awareness of each other's position through communication channels. They are also aware when teammates have detected the Red evader because this information is also communicated to each other. Blue team does not always have knowledge of Red position so at training time, this information must be hidden at training time if unknown to the Blue agents.

6.1.3 Situation and Action Vector specification

Once the training logs were obtained, the elements of the situation vector S were defined. These data consists of information that is available in the simulation log and at run-time. Since the ships have communication capability, the position of all Blue ships is always known and can be part of the situation vector. Once the Red evader's position is detected, it is known to all Blue ships. However, since Red position is not always known a flag indicating its presence is used in the situation vector. All of these elements can be found or derived from the log file and are available via messages or sensors to the run-time agents. Table 6-4 lists the data elements contained in S and their types.

Table 6-4 Pursuit-Evasion Situation Vector Data Elements

Element Name	**Description**	**Data Type**
myState	Enumeration indicating current context of the agent	Context
myType	Enumeration indicating numeric assignment of the team member, i.e., BLUE1, BLUE2, BLUE3, BLUE4	PlayerType
myPosition	This team member's current position in x,y	Position
Blue1Position	BLUE1's current position in x,y	Position
Blue2Position	BLUE2's current position in x,y	Position
Blue3Position	BLUE3's current position in x,y	Position
Blue4Postion	BLUE4's current position in x,y	Position
redPositionKnown	Boolean indicating whether the agent knows the position of the Red evader.	boolean
RedPosition	If known, the position of the red evader in x,y	Position
TeamContext	Enumeration indicating the current team context	TeamContext

The action vector associated with the situation vector in the behavior map for each context will contain from one to three possible actions. The one action that will always be present is a movement action. The possible values for this action are: UP, DOWN, LEFT, RIGHT and NONE. The second possible action is a context transition. The third possible action is SEND_MESSAGE. This action sends a message with the detected Red position to all the blue ships. The enumerations for all of these actions are available in Table 6-5. Table 6-5 also contains the other enumerations used in the simulation to describe individual contexts, player types, and team contexts. As in the bucket brigade prototype, the situation vector is implemented as a Java class named Situation and the enumerations are available in a class named Constants. The COLTS code files developed for the pursuit-evasion game are available in Appendix C.

Table 6-5 Enumerations

Enumeration	Values	Description
Context	SEARCH_TOP_LEFT, SEARCH_TOP_RIGHT, SEARCH_BOT_LEFT, SEARCH_BOT_RIGHT, GO_TO_TOP, GO_TO_RIGHT, GO_TO_LEFT, GO_TO_BOTTOM, INTERCEPT_TOP, INTERCEPT_BOTTOM, INTERCEPT_RIGHT, INTERCEPT_LEFT	The names of the various individual contexts in the system.
Actions	UP, DOWN, RIGHT, LEFT, NONE, NOTIFY_RED_POSITION, TRANS_SEARCH_TOP_LEFT, TRANS_SEARCH_TOP_RIGHT, TRANS_SEARCH_BOT_RIGHT, TRANS_SEARCH_BOT_LEFT, TRANS_GO_TO_TOP, TRANS_GO_TO_RIGHT, TRANS_GO_TO_LEFT,TRANS_GO_TO_BOTTOM, TRANS_INTERCEPT_TOP, TRANS_INTERCEPT_BOTTOM, TRANS_INTERCEPT_RIGHT, TRANS_INTERCEPT_LEFT	Possible actions for an agent to take within a time-step. Enumerations beginning with TRANS indicate that an action is a context switch.
TeamContexts	SEARCHING, BOXING	The names of the team contexts.
Player Types	BLUE1, BLUE2, BLUE3, BLUE4, RED	Enumerations assigned to each player in the game by the game engine. BLUE# are team members, RED indicates the red evader.

6.1.4 Contextualization

As with the bucket brigade, a contextual analysis of the observed game is necessary in order to set up the training algorithm for the COLTS pursuit team. The contextual analysis was done manually by comparing the available log data with the behavior observed by watching the observed team. The observed team implemented four distinctly different agents to represent the teams.

The team has two easily observed contexts. The first is Search, where the four pursuers are attempting to find the location of the Red evader. In the observed team, each of the four pursuers has an assigned quadrant to search in a predefined pattern until someone detects the Red evader. Once the Red evader has been located, the team context switches to Capture context. In this context, each of the four pursuers attempts to intercept the Red evader and block a particular side. Figure 6-4 shows the two team contexts. Note that arrow goes only from Search to Capture and not back. Once the observed team had a location, the observed detecting ship always stayed within range of the evader so there was no need to return to a search pattern.

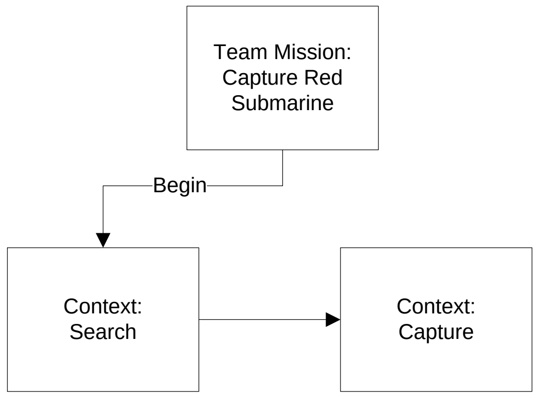


Figure 6-4 Pursuers Team Contexts

The individual contexts for each ship are quite similar to each other, but have important differences. The search patterns are similar, but in vastly different areas for each of the four quadrants so that separate contexts are needed for each search quadrant. In addition, each blue ship has a predetermined position around the evader and intercepting that location is slightly different from intercepting the position of the evader only. Therefore, intercept contexts must also be different for each ship.

Each ship begins the mission in a search context. There are four different search contexts, one for each quadrant to be searched. When the position of the pursued Red submarine is detected by one or more of the Blue ships, all ships are notified of the location. Once the position is known, the ships proceed to transition to either an intercept context or a blocking context. An intercept context is used when the evader is outside sensor range of the ship and the ship needs to intercept its position using the information provided by the closer ship or ships. A blocking context is when the submarine is within sensor range and the ship must get to a particular side of the submarine to block and capture it.   Figure 6-5, Figure 6-6, 6-7 , and 6-8 show the contexts and their names of each of the blue ships. In total, there are two team contexts and twelve individual contexts of the blue team.

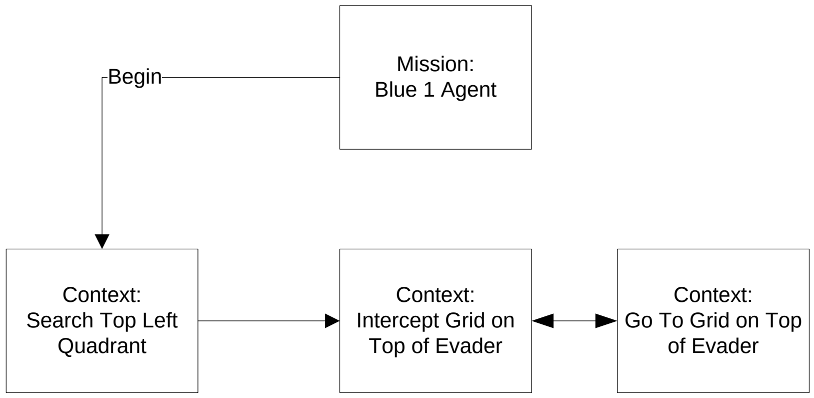


Figure 6-5 Blue Ship 1 Individual Contexts

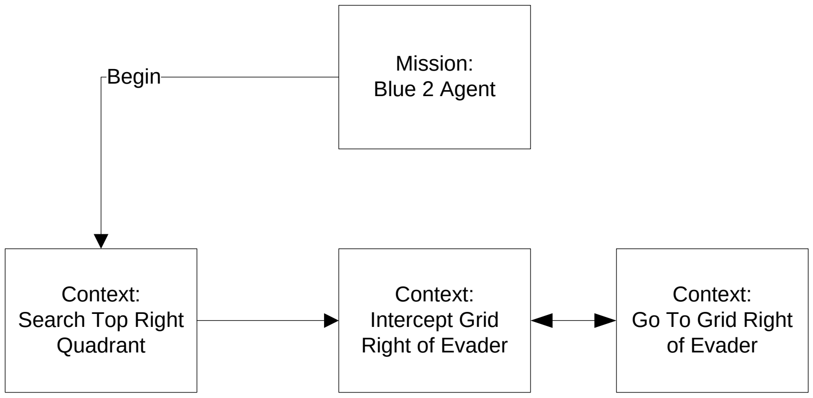

Begin
Mission:
Blue 2 Agent
Context:
Search Top Right Quadrant
Context:
Intercept Grid Right of Evader
Context:
Go To Grid Right of Evader

Figure 6-6 Blue Ship 2 Individual Contexts

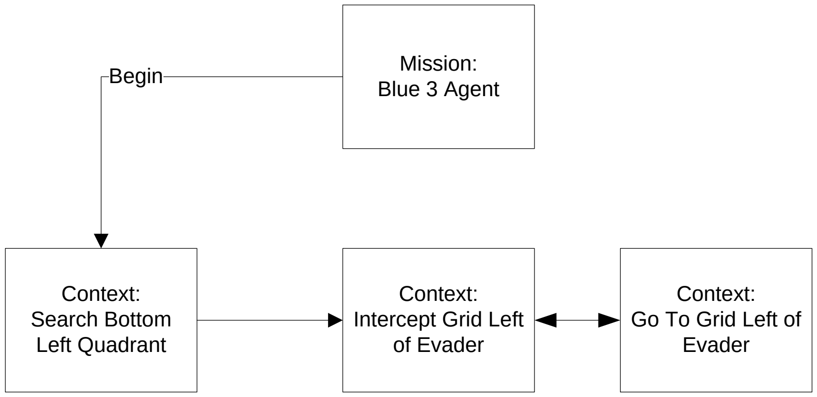

Begin
Mission:
Blue 3 Agent
Context:
Search Bottom Left Quadrant
Context:
Intercept Grid Left of Evader
Context:
Go To Grid Left of Evader

Figure 6-7 Blue Ship 3 Individual Contexts

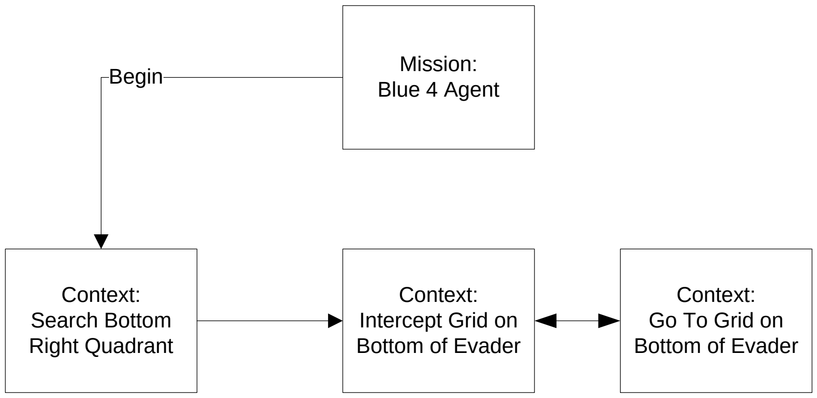

Begin
Mission:
Blue 4 Agent
Context:
Search Bottom Right Quadrant
Context:
Intercept Grid on Bottom of Evader
Context:
Go To Grid on Bottom of Evader

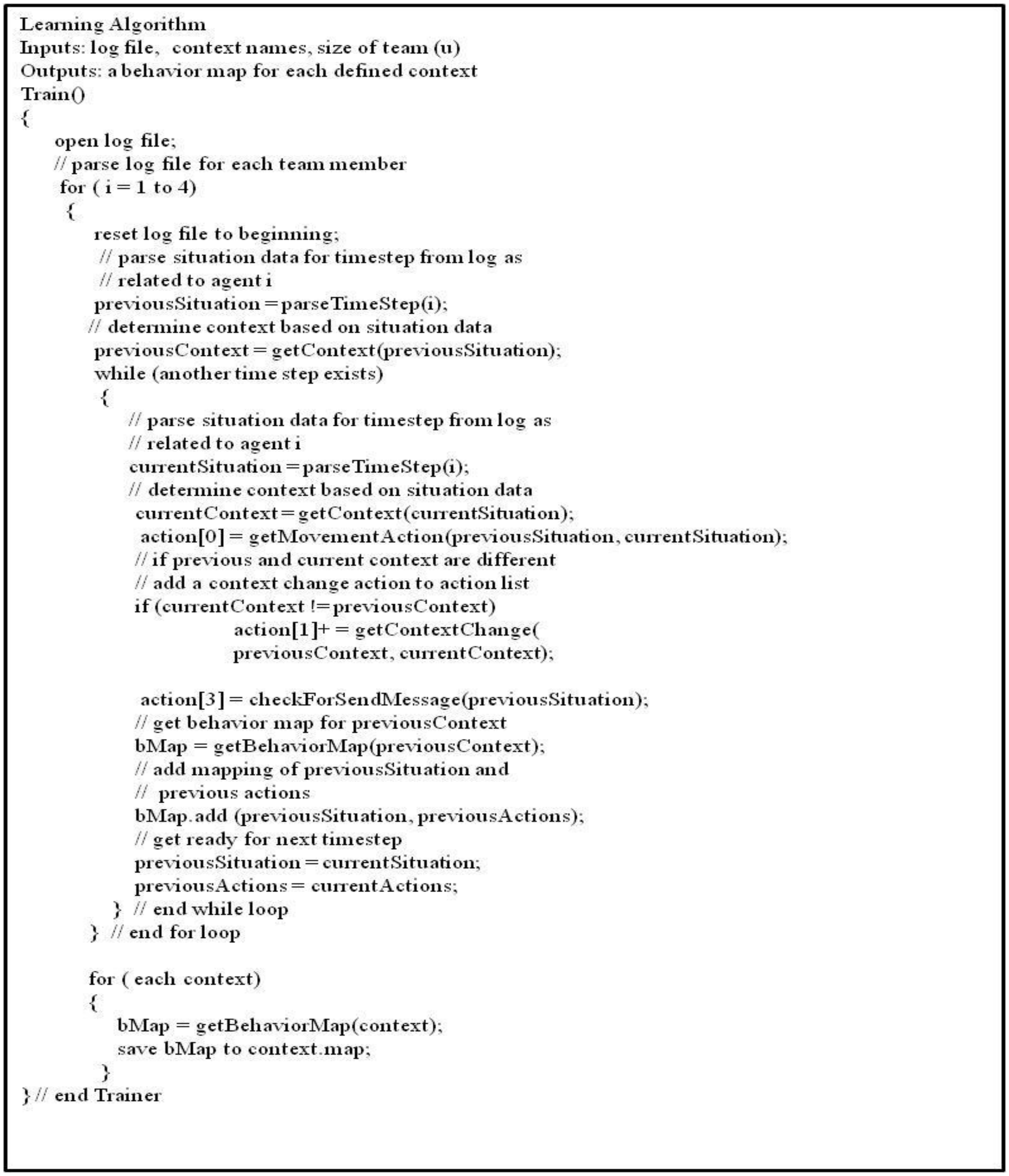
Figure 6-8 Blue Ship 4 Individual Contexts

6.2 Learning Module

Once the situation vector, action vector and contexts were determined, the development of the Trainer application for the pursuit game could begin. The stand-alone Java Trainer application developed for the bucket brigade prototype was used as a starting point. However, since this domain has a differently formatted log file and new contexts, the code to parse the log file and determine context had to be modified. The parsing code was modified to read the pursuit-evasion data log and convert the data parsed into the new Situation class. Context was determined by two basic pieces of data. The data elements used to determine context for each Blue ship are *redPositionKnown* and *redPosition*.

The SEARCH context is always the starting context at run-time. If Red position is known and the ship is more than five steps from the Red evader, the ship is in the appropriate INTERCEPT context for the ship number. If Red position is known and ship is less than five steps from the Red evader, the ship is in the appropriate GO_TO context for the player number. Figure 6-5, 6-6, 6-7 and 6-8 above show the mapping of the different categories of context to ship number.

The actions could not be directly parsed from the log, so all had to be determined from the changes in one time-step to the next. However, the basic algorithm was the same as the conceptual algorithm in Chapter 4. Figure 6-9 updates the learning algorithm presented in Chapter 4 to show the prototype 2 specific needs and the following section describes the algorithm.



```
Learning Algorithm
Inputs: log file,  context names, size of team (u)
Outputs: a behavior map for each defined context
Train()
{
     open log file;
     // parse log file for each team member
      for ( i = 1 to 4)
       {
           reset log file to beginning;
            // parse situation data for timestep from log as
            // related to agent i
           previousSituation = parseTimeStep(i);
          // determine context based on situation data
           previousContext = getContext(previousSituation);
           while (another time step exists)
            {
                 // parse situation data for timestep from log as
                 // related to agent i
                 currentSituation = parseTimeStep(i);
                 // determine context based on situation data
                  currentContext = getContext(currentSituation);
                   action[0] = getMovementAction(previousSituation, currentSituation);
                  // if previous and current context are different
                  // add a context change action to action list
                  if (currentContext != previousContext)
                                  action[1]+ = getContextChange(
                                 previousContext, currentContext);

                   action[3] = checkForSendMessage(previousSituation);
                  // get behavior map for previousContext
                  bMap = getBehaviorMap(previousContext);
                  // add mapping of previousSituation and
                  //  previous actions
                  bMap.add (previousSituation, previousActions);
                  // get ready for next timestep
                  previousSituation = currentSituation;
                  previousActions = currentActions;
              }  // end while loop
          }  // end for loop

          for ( each context)
          {
               bMap = getBehaviorMap(context);
               save bMap to context.map;
           }
} // end Trainer
```

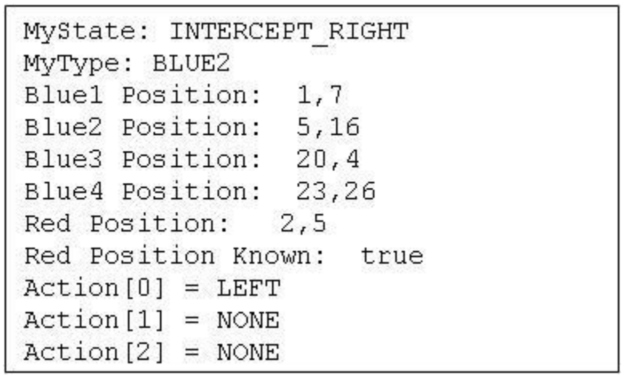
Figure 6-9 Pursuit Team Learning Algorithm

6.2.1 Learning Algorithm Description

As in the bucket brigade prototype, the log file is parsed separately for each of the observed team members. Since the observed team has four members, the log file was parsed and analyzed four times. For each time-step, the log file is read and a Situation class for time, t, is created. Before actions are assigned to this Situation instance, the Situation class for time, t+1, is created and its context determined. The Situation instances for time t and time t+1 are compared. First, the position data for the ship being analyzed is compared to the previous time-step to determine the movement action taken in time t. If the *x* coordinate of the ship has changed positively, the movement action is RIGHT. If the *x* coordinate has changed negatively, the movement action is LEFT. If the *y* coordinate has changed positively, the movement action is DOWN. If the *y* coordinate has changed negatively, the movement action is UP. If no coordinates changed, the movement action is NONE.

Next, the contexts of the two Situation classes are compared. If they are different, an appropriate context transition action is added to the action set. Finally, if the agent being analyzed is within sensor range of the Red agent, a SEND_MESSAGE action is added to the action set. This action set is paired with the Situation instance for time t. This mapping of Situation class to actions is stored in the behavior map for the context of the Situation instance at time t. As in the bucket brigade prototype, this behavior map is implemented as a Java Hashmap with the Situation class used as a key. When parsing of the input log file is complete for all agents, the behavior maps are stored as binary files for use by the run-time COLTS agents.

While the behavior maps have a similar format to those of the bucket brigade, the data stored are different. An example of an entry in the INTERCEPT_RIGHT context behavior map is shown in Figure 6-10.



```
MyState: INTERCEPT_RIGHT
MyType: BLUE2
Blue1 Position:  1,7
Blue2 Position:  5,16
Blue3 Position:  20,4
Blue4 Position:  23,26
Red Position:   2,5
Red Position Known:  true
Action[0] = LEFT
Action[1] = NONE
Action[2] = NONE
```

Figure 6-10 Example Behavior Map

6.3 Run-time Module

The COLTS agent framework developed for the bucket brigade provided an excellent starting point for prototype 2. Since both are time-stepped simulations, the Agent and Context classes required little change. The Agent class required some update to acquire the situational data from the available sensors before calling the current context and a change to the processing of data received as a message from a teammate. Within the team context, the instantiations of the various contexts were tailored to the pursuit-evasion game contexts. Figure 6-11 shows a UML class diagram of the prototype run-time application. In Figure 6-12, the replacement of the observed game agents with the COLTS agents is shown in block diagram.

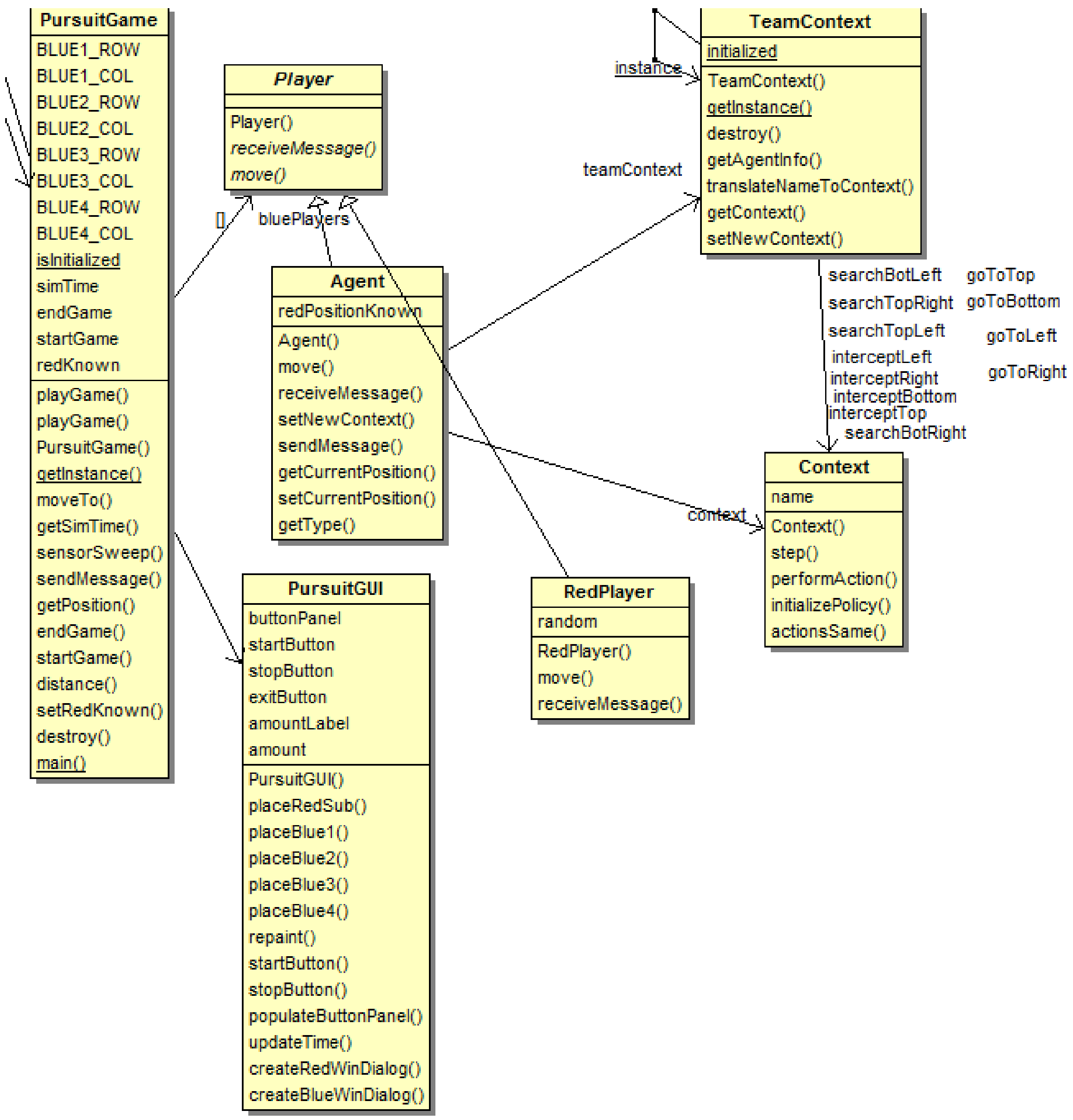


Figure 6-11 COLTS Prototype Run-time UML

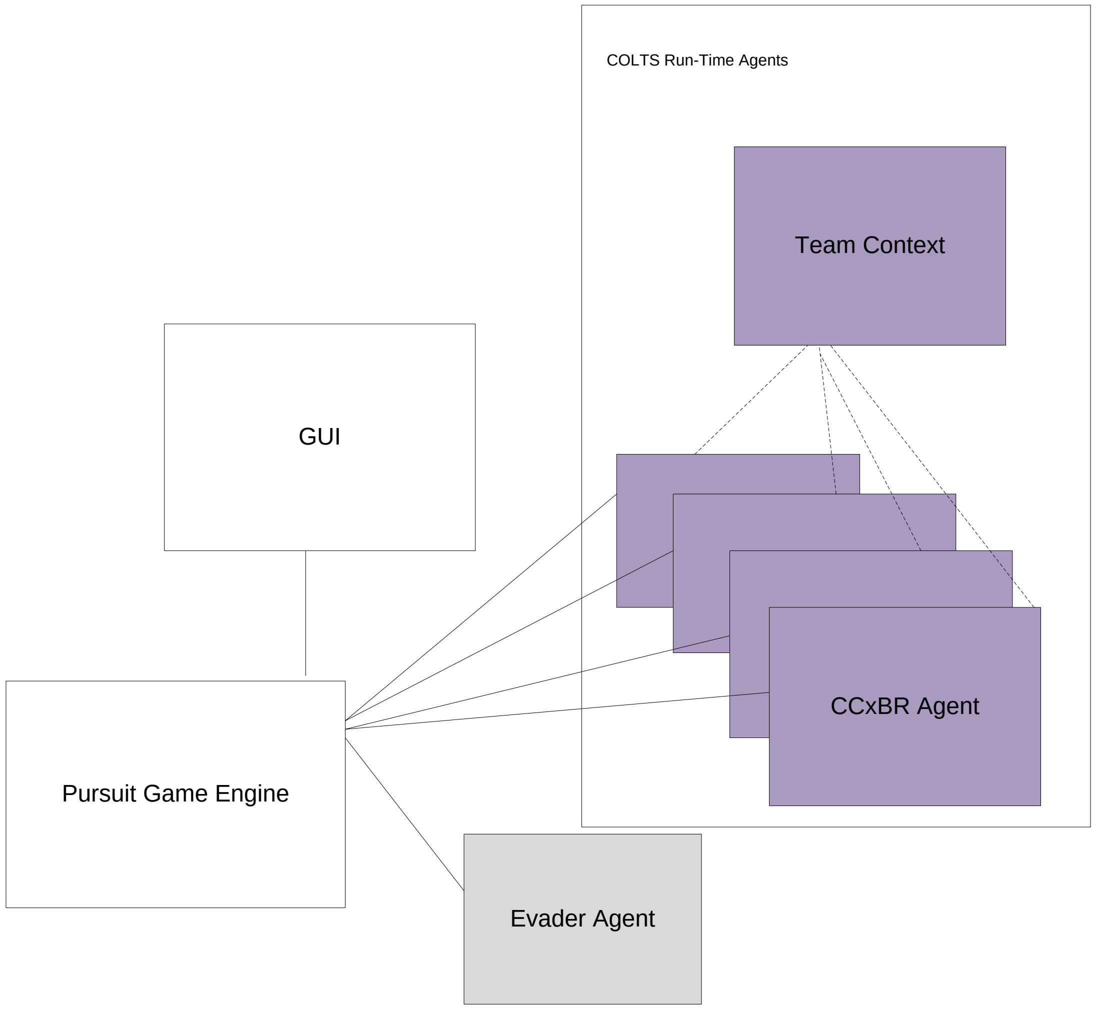


Figure 6-12 COLTS Prototype Pursuit-Evasion Game

6.3.1 Contextually-weighted k-Nearest Neighbor Algorithm

Once the behavior maps were populated by the training algorithm and the run-time agents in place, the next step was to determine the weights for the contextually-weighted k-nearest-neighbor algorithm. Although there are a total of 12 possible contexts for the agents, the contextual weighting is divided into three categories. The first category is the search category and includes the contexts named SEARCH_TOP_RIGHT, SEARCH_TOP_LEFT, SEARCH_BOT_RIGHT and SEARCH_BOT_LEFT. In these contexts, the position of the submarine is unknown and the ships are following a predetermined search pattern within a particular context to find it.

When creating the initial weights for this category of contexts, the current position of the ship is the most important factor in determining the next move. The positions of the remaining three ships factor slightly into the decision, and only when they may be blocking the desired move of the ship. The position of the Red evader is unknown and therefore receives a weight of zero in this category. Whether or not the position of the Red evader is known is also highly weighted as this will determine whether a context switch is needed. After some brief experimentation with various weights, a set of weights were calculated and Table 6-6 shows the compared data and associated weights used in this category of contexts. The table shows two possible values for each Blue ship. The first value reflects the weight used when the value of *myType* matches the designation of the ship building the weight vector is that ship. The second value reflects the weight when the designated ship is a teammate. For example, if the ship building the weight vector is designated BLUE1, the BLUE1 position weight difference will be 3.0. The weights for BLUE2, BLUE3 and BLUE4 will all be 0.25.

Table 6-6 Search Context Weights

Data Description	**Weight**	**Explanation**
Blue1 Position Diff	3 or 0.25	Weight = 3 if this is Blue1, 0.25 otherwise
Blue2 Position Diff	3 or 0.25	Weight = 3 if this is Blue1, 0.25 otherwise
Blue3 Position Diff	3 or 0.25	Weight = 3 if this is Blue1, 0.25 otherwise
Blue4 Position Diff	3 or 0.25	Weight =3 if this is Blue1, 0.25 otherwise

The second category of contexts is the intercept category. The names of the context associated with this category are INTERCEPT_TOP, INTERCEPT_LEFT, INTERCEPT_RIGHT and INTERCEPT_BOTTOM. In this category, the ships are attempting to move to a position that will allow them to capture the Red submarine. A ship in an intercept context is receiving the Red position from another ship as the position is outside sensor range. The initial weights for this category were set to emphasize the location of this ship and the Red submarine. However, it soon became clear during testing of the algorithm that this was not necessarily calling for an action that best reflected the behavior of the observed team. A modification to the algorithm to compute the angle between the ship and the Red position and giving a strong weight to that value proved to be a better comparison. Table 6-7 shows the calculation and weights used to determine a nearest-neighbor score for these contexts. While it appears the position of the ship is given no weight, this position is used in the calculation of the angle to Red evader and distance to Red evader. In this context, the position of the ship in relation to the Red evader is much more important than actual position on the grid.

Table 6-7 Intercept Context Weights

Data Description	Weight	Explanation
Blue1 Position Diff	0.0	Not used in these contexts.
Blue2 Position Diff	0.0	Not used in these contexts.
Blue3 Position Diff	0.0	Not used in these contexts.
Blue4 Position Diff	0.0	Not used in these contexts.
Distance to Red Diff	1.0	Calculated from current position and Red position provided in Situation data.
Angle to Red Diff	3.0	Calculated from current position and Red position provided in Situation data

The third and final category of the contexts is the capture category. In this category, the associated contexts are GO_TO_TOP, GO_TO_LEFT, GO_TO_RIGHT, and GO_TO_BOTTOM. In this category, the ship is within sensor range of the red submarine and is trying to capture and maintain a particular position around the Red submarine. This was the most difficult category of context to weight. Initially, it was weighted similarly to the intercept category. However, this did not account for the type of movement that the observed agents would perform to maneuver around teammates. It became necessary to look at not only the angle to the Red evader, but also the distance. Since teammates can be in close proximity and cause roadblocks, the angles and distances between the various blue ships were also taken into account to find the situation-action pair that accurately reflected the behavior of the team mates. Table 6-8 shows the calculation and weights used to determine nearest-neighbor scores in these contexts.

Table 6-8 Go_to Context Weights

Data Description	**Wt**	**Explanation**
Blue1 Position Diff	0.0	Not used in these contexts.
Blue2 Position Diff	0.0	Not used in these contexts.
Blue3 Position Diff	0.0	Not used in these contexts.
Blue4 Position Diff	0.0	Not used in these contexts.
Distance to Red Diff	1.0	Calculated from current position and Red position provided in Situation data.
Angle to Red Diff	3.0	Calculated from current position and Red position provided in Situation data
Distance to Blue1 Diff	0.2	Calculated from current position and blue position data provided in Situation data
Distance to Blue2 Diff	0.2	Calculated from current position and blue position data provided in Situation data
Distance to Blue3 Diff	0.2	Calculated from current position and blue position data provided in Situation data
Distance to Blue4 Diff	0.2	Calculated from current position and blue position data provided in Situation data
Angle to Blue1 Diff	1.5	Calculated from current position and blue position data provided in Situation data
Angle to Blue2 Diff	1.5	Calculated from current position and blue position data provided in Situation data
Angle to Blue3 Diff	1.5	Calculated from current position and blue position data provided in Situation data
Angle to Blue4 Diff	1.5	Calculated from current position and blue position data provided in Situation data

6.4 Results and Conclusions

This section describes the results from the experimentation performed using the COLTS pursuit team. The criteria used to evaluate the COLTS team are discussed. In addition, modifications to the original run-time algorithm are discussed and final results are analyzed.

6.4.1 Criteria for Evaluation

The question arises as to what measurable statistic is a true indicator of the behavior of the team. Some measure is needed to determine whether or not the COLTS prototype exhibits the same team effectiveness as the observed team. The win ratio at a variety of different starting position was chosen as a measure of effectiveness. The decision to use this particular statistic was based on comparison of two existing teams with different algorithms used for the pursuit team. Comparison of the two existing teams indicated that the win ratios are affected by the algorithm used by the pursuit team. To prove this theory, ten test cases used to test an expert system implementation of the pursuit game by Proenza in 1997 [Proenza, 1997] were duplicated by the observed team. While the basic game was the same, Proenza took a different approach to the problem. He used a central controller to direct the four ships that were implemented in an expert systems tool known as CLIPS. Another difference was that his ships always knew where the Red evader was, but not necessarily each other. The Blue ships only knew their teammates position if a collision was possible.

The comparison between Proenza's pursuit team and the observed pursuit team used for this dissertation is presented in Table 6-9 Test Case Comparison. Proenza used a similar random evader, but calculated three possible outcomes; win, lose and stalemate. A stalemate was declared when the pursuers were able to get three sides captured and close to an edge, but not the fourth. The table shows the Blue win percentage for Proenza's team when the Blue team predominantly won. Otherwise, Red Escape or Stalemate is listed indicating that the Red evader usually won. In the observed team's implementation, that is declared a loss as the Red evader would eventually move to the opposite edge. [Proenza, 1997] It quickly becomes obvious from the comparison that these are very different implementations.

The results for the various starting configurations are almost opposite; test cases which caused Proenza's prototype to favor Red wins or stalemates have very high Blue win ratios in the new system. The two test cases of Proenza with high Blue win ratios have poor showings in the new system. This is probably because the new system had to search for the Red evader position initially. The results of this comparison show that two different algorithmic approaches to the same problem can have drastically different win percentages. Therefore, if the COLTS prototype has win ratios comparable to the observed team, it can be considered to be properly imitating the same algorithm as the observed team.

Table 6-9 Test Case Comparison

Proenza Test Case #	Observed Team Test Case #	Proenza Result (blue win %)	Observed Team Result (blue win %)
2	2	Red Escape	88%
28	18	86%	41%
30	19	Stalemate	59%
29	20	92%	72%
8	21	78%	67%
10	22	Stalemate	83%
12	23	Red Escape	94%
13	24	Stalemate	47%
27	27	74%	79%
23	28	Stalemate	97%

Once the win ratio was chosen as a test criterion, a series of games was run with thirty different starting positions for the blue agents. Each series of games consisted of 1000 games. These games serve as a baseline against which the COLTS prototype can be compared. For each starting position, a set of 10 series was run to determine the average win ratio for that starting position. So a total of 10000 games were played at each starting position to get the average Blue win ratio for the observed team.

6.4.2 Initial Results

In the bucket brigade prototype, the contextually-weighted k-nearest-neighbor algorithm assumed that k would always equal one meaning only one low-scoring match for any given situation within the behavior map. This assumption yielded good results in the pursuit game, but not good enough. Table 6-10 shows the observed team and the initial COLTS team average win percentage in a series of 1000 games. Results show that the win percentages between COLTS and the observed team are close, but not close enough to be able to say statistically that they are similar. The COLTS team generally has a lower win ratio than the observed team, but in a few cases there were actually test cases that had a much higher win ratio. The goal is to be able to perform a chi-square goodness-of-fit statistical test and have the resulting value show the results to be statistically similar within a 95% confidence interval. The chi-square statistic is computed for each test case win ratio and shown in Table 6-10. The last column of the table indicates whether the COLTS team win ratio is statistically similar to the observed team. Table 6-11 shows the 2x2 contingency table used to compute the chi-square statistic.

Table 6-10 Initial Results with low-score nearest-neighbor algorithm

Test Case	Observed Win %	COLTS Win %	Chi-square Statistic	Match?
TC1	72.3	67.0	0.660	TRUE
TC2	88.3	74.9	12.350	FALSE
TC3	64.0	61.5	0.129	TRUE
TC4	58.7	45.7	3.413	TRUE
TC5	48.2	50.4	0.099	TRUE
TC6	67.5	59.6	1.350	TRUE
TC7	68.7	65.6	0.190	TRUE
TC8	70.1	64.4	0.737	TRUE
TC9	70.1	64.8	0.637	TRUE
TC10	69.6	61.8	1.343	TRUE
TC11	68.9	58.4	2.369	TRUE
TC12	63.2	54.6	1.546	TRUE
TC13	15.7	8.9	1.171	TRUE
TC14	97.5	92.2	2.835	TRUE
TC15	54.0	44.2	1.937	TRUE
TC16	79.7	62.3	6.691	FALSE
TC17	66.7	71.0	0.437	TRUE
TC18	41.6	14.0	18.976	FALSE
TC19	60.0	48.7	8.000	FALSE
TC20	72.7	89.0	8.544	FALSE
TC21	67.7	64.0	2.564	TRUE
TC22	83.2	60.8	12.407	FALSE
TC23	94.6	82.9	6.870	FALSE
TC24	47.3	57.6	2.124	TRUE
TC25	17.2	19.6	0.189	TRUE
TC26	81.2	62.5	8.655	FALSE
TC27	79.7	62.3	7.361	FALSE
TC28	98.0	95.7	0.832	TRUE
TC29	76.8	58.2	7.867	FALSE
TC30	80.5	61.6	8.693	FALSE

Table 6-11 Chi-square 2x2 Contingency Table

	Observed Game	**Prototype Game**	**Totals**
Win Percentage	a	b	a+b
Loss Percentage	c	d	c+d
Totals	a+c	b+d	a + b + c + d = N

The chi-square statistic was calculated based on the values in the table using the following formula:

$$\chi^2 = \frac{(ad-bc)^2 * (a+b+c+d)}{(a+b)(c+d)(b+d)(a+c)} \qquad \textbf{(6-1)}$$

In order for the two win ratios to be declared statistically similar within a 95% confidence interval, the chi-square statistic must be less than 3.841. This particular statistic is derived from a chi-square table indicating 95% confidence interval. The results are generally good, but over a third of the test cases do not pass the chi-square statistical test and only a few come within a 99% confidence interval. (This is defined as a chi-square statistic of less than 2.706.)

In order to improve these statistics and meet the goal of all 30 test cases passing the chi-square test within a 95% confidence interval, additional weight adjustments were tested. However, these only had the effect of switching which test cases passed the test and did not improve the overall pass/fail ratio. Testing showed that the majority of problems came in the GO_TO contexts. Through additional testing, it became clear that in some cases it was possible for more than one situation to match the low-score criteria. This led to the addition of additional run-time logic to the contextually-weighted k-nearest-neighbor algorithm to handle a k value greater than one.

6.4.3 Mismatch Logic

The reason for the large value of k in the nearest-neighbor algorithm is that in some contexts, the actual positions of the ships and submarine are not relevant; it is their relation to each other that affects the score. Tracing the code with a debugging software made it clear that k was often greater than 1. So mismatch logic was implemented as part of the contextually-weighted nearest neighbor algorithm.

A running count of situations with the same score is kept. Should differing actions be suggested by the situations with matching score, the actions with the most situations is chosen rather than the first one encountered. The addition of this logic and some additional weight tuning resulted in the achievement of passing the chi-square test on all thirty test cases. Weight tuning was performed by using a subset of the test cases with a limited number of test runs. Early on it became clear that certain test cases were more difficult to duplicate. A total of five test cases were used. Test Cases 1, 5, 19, 25 and 26 were the test cases used. Test Cases 1 and 5 were from the trained test cases and used as control test cases to make sure weight changes didn't disrupt trained behavior. The others were chosen because they were the furthest from passing the chi-square test with a 95% confidence interval and showed the most sensitivity to changing weight vectors. Other test cases did not show as much variation in win ratios when weight vectors were altered.

This indicates that the situations in those test cases were probably very close to the situations defined in the behavior maps so the best match was made unless weight vectors were drastically changed. Because the variations in the weight vectors used for experimentation were quite small (typically less than 1.0), they did not impact these well-represented test cases. The weights chosen gave the best overall results rather than any one test case.

Final Results

Table 6-12 shows the resulting win ratios and the chi-square statistics after the logic to handle k values greater than one was added. In addition to all of the test cases matching the observed

Table 6-12 Final Prototype Win Percentages

Test Case	Observed Win %	COLTS Win %	Chi-square statistic	Match?
TC1	72.3	68.9	0.275	TRUE
TC2	88.3	87.8	0.019	TRUE
TC3	63.9	62.9	0.024	TRUE
TC4	58.7	57.3	0.033	TRUE
TC5	48.2	51.4	0.207	TRUE
TC6	67.5	66.5	0.023	TRUE
TC7	68.7	67.9	0.015	TRUE
TC8	70.1	70.0	0.000	TRUE
TC9	70.0	69.9	0.000	TRUE
TC10	69.6	65.8	0.327	TRUE
TC11	68.9	65.5	0.258	TRUE
TC12	63.2	60.9	0.117	TRUE
TC13	15.7	21.2	0.986	TRUE
TC14	97.5	97.5	0.000	TRUE
TC15	54.0	52.2	0.068	TRUE
TC16	79.7	79.0	0.016	TRUE
TC17	66.7	62.2	0.436	TRUE
TC18	41.6	44.8	0.340	TRUE
TC19	60.0	61.1	0.026	TRUE
TC20	72.7	79.1	1.106	TRUE
TC21	67.7	67.4	0.000	TRUE
TC22	83.2	74.1	1.730	TRUE
TC23	94.6	88.3	2.540	TRUE
TC24	47.3	46.7	0.007	TRUE
TC25	17.2	27.9	3.264	TRUE
TC26	81.2	82.5	0.056	TRUE
TC27	79.7	79.1	0.011	TRUE
TC28	98.0	98.0	0.000	TRUE
TC29	76.8	77.7	0.024	TRUE
TC30	80.5	79.9	0.012	TRUE

within a 95% confidence level, two-thirds of the test cases match with a 99% confidence level. It is interesting to note that test case 25 that was closest to failing the chi-square test, does so by having a higher win ratio than the observed. During weight testing, it became clear that during test cases 25 and 19, the observed team made some diametrically opposing choices given similar situations. Weights that have test case 25 passing with a win percentage much closer to the observed would cause test case 19 to fail. Finally, a set of weights was found that allowed both to pass, but test case 25 still comes very close to failing. The weights shown in Table 6-6, Table 6-7, and Table 6-8 reflect the final weights used.

In order to test the generalization of the behavior of the prototype, an additional set of test runs was made using the same *smart* evader used to test the observed team. The training cases and prior runs were all done using a totally random evader to represent the Red submarine. The smart evader representing the Red submarine is also random until it is detected by one of the ships. Once detected, the evader will aim directly for the nearest edge. In the observed game, this still resulted in an nearly 0% win percentage for the blue ship so some additional randomness was added when the smart evader is heading for the edge. The prototype was not trained with any examples of the observed playing this smarter evader so it is an excellent test of generalization to see if it can match the win percentages of the observed team against this new opponent.

As shown in Table 6-13, the results were not quite as good as against the random evader, but 28 of the 30 test cases still pass the chi-square goodness-of-fit statistical test within a 95% confidence interval. The two test cases that fail both do so with a significantly higher win percentage so in those two test cases, the prototype performed better than the observed. This was not the trend in all test cases.

Table 6-13 Smart Evader Test Results

Test Case	Observed Win %	COLTS Win %	Chi-square statistic	Match?
TC1	20.4	18.0	0.183	TRUE
TC2	59.2	53.6	0.638	TRUE
TC3	19.3	15.6	0.475	TRUE
TC4	13.2	13.9	0.019	TRUE
TC5	20.4	14.4	1.272	TRUE
TC6	22.6	17.0	1.001	TRUE
TC7	23.2	19.4	0.429	TRUE
TC8	17.7	13.8	0.568	TRUE
TC9	24.3	22.1	0.142	TRUE
TC10	18.1	15.8	0.189	TRUE
TC11	16.4	11.8	0.891	TRUE
TC12	14.8	12.9	0.147	TRUE
TC13	3.5	4.2	0.062	TRUE
TC14	75.0	75.0	0.0001	TRUE
TC15	20.4	18.4	0.133	TRUE
TC16	37.2	25.3	3.207	TRUE
TC17	23.3	15.9	1.751	TRUE
TC18	9.2	6.5	0.567	TRUE
TC19	11.7	30.2	10.268	FALSE
TC20	37.1	30.9	0.859	TRUE
TC21	23.8	19.8	0.465	TRUE
TC22	32.6	33.6	0.022	TRUE
TC23	57.4	46.9	2.235	TRUE
TC24	13.0	24.6	4.416	FALSE
TC25	7.6	9.1	0.151	TRUE
TC26	21.7	20.1	0.079	TRUE
TC27	30.1	26.9	0.254	TRUE
TC28	77.6	75.0	0.183	TRUE
TC29	42.0	35.7	0.843	TRUE
TC30	28.8	23.6	0.694	TRUE

6.4.4 Conclusions

The results of prototype 2 show that behavior maps combined with a contextually-sensitive selection algorithm can effectively transfer collaborative behavior from one source to another with a team size of 4. The approach was verified through 30 test cases. A test case was defined by the starting position of the four blue ships comprising the pursuit team. The comparison of the COLTS team to the observed team was done by comparing the win percentage for each test case. The win percentage was determined from the results of a test run of 1000 games in each test case. Comparing the win percentages of the observed team to previous implementations in literature using different approaches showed that a different implementation would result in a different win percentage. The win percentage of each test case of the prototype was statistically compared to the win percentage of the observed team using a chi-square goodness-of-fit algorithm. The goal was to declare the prototype's win percentage the same as the observed with a 95% confidence interval. A comparison of the results for each test case is shown in Figure 6-13. A similar comparison of the results from the smart evader testing is shown in Figure 6-14.

The training algorithm created behavior maps from a log of 150 games using 15 different test cases. The contextually-weighted k-nearest-neighbor algorithm had its roots in k-nearest neighbor algorithms, but evolved into a more sophisticated selection algorithm by utilizing statistical techniques reminiscent of Gaussian mixture models. The initial approach was to use

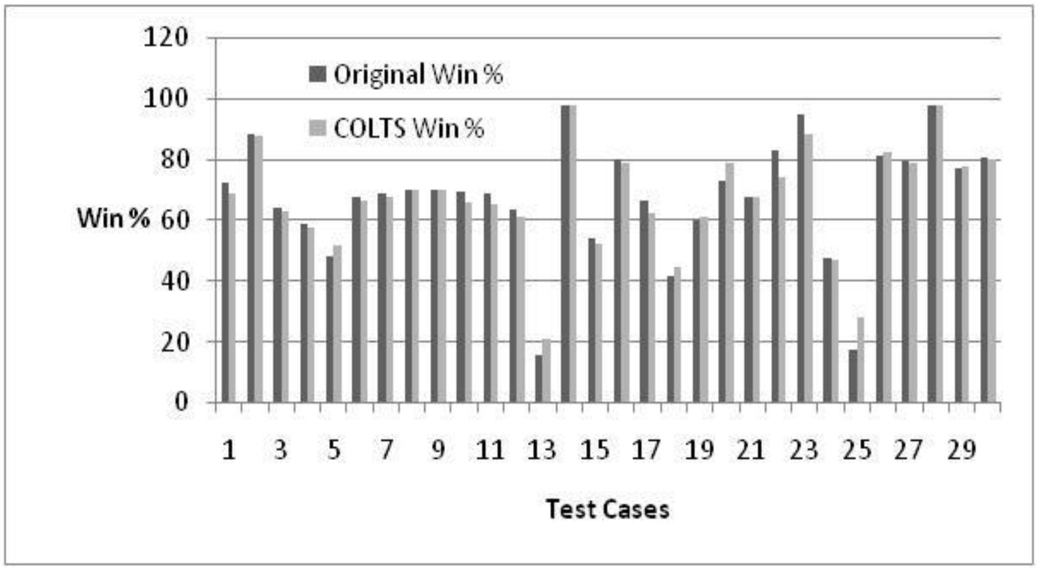


Figure 6-13 Comparison of Results for Random Evader

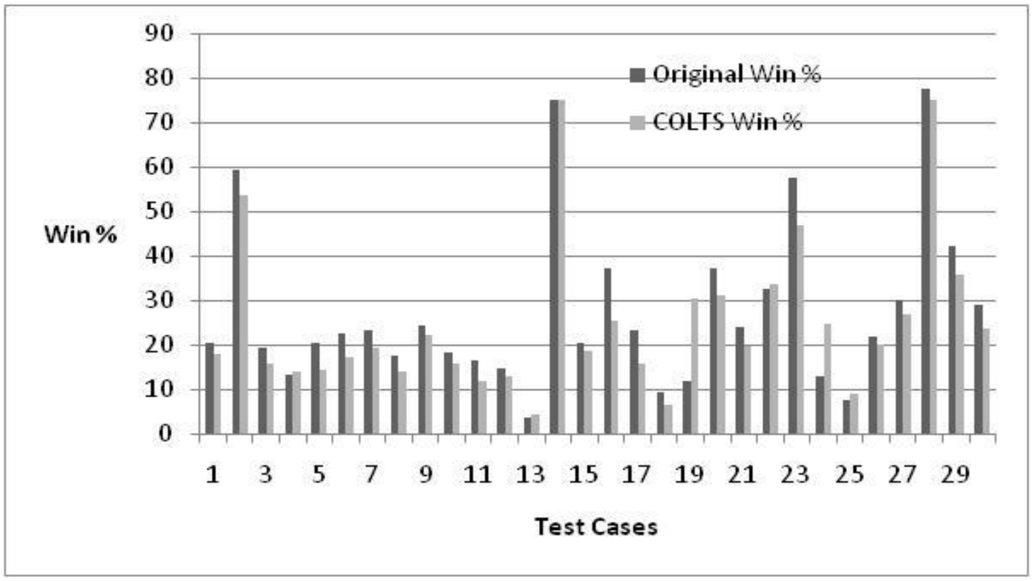


Figure 6-14 Comparison of Results for Smart Evader

Table 6-14 Chi-square Statistic Comparison

Test Case	First Draft Chi-square Statistic	Final Chi-square Statistic	Difference
TC1	0.6600	0.2751	-0.3849
TC2	12.3500	0.0190	-12.3310
TC3	0.1290	0.0240	-0.1050
TC4	3.4130	0.0330	-3.3800
TC5	0.0986	0.2070	0.1084
TC6	1.3500	0.0230	-1.3270
TC7	0.1900	0.0150	-0.1750
TC8	0.7370	0.00002	-0.7369
TC9	0.6370	0.00008	-0.6369
TC10	1.3430	0.3270	-1.0160
TC11	2.3690	0.2580	-2.1110
TC12	1.5460	0.1170	-1.4290
TC13	1.1711	0.9860	-0.1851
TC14	2.8350	0.0002	-2.8348
TC15	1.9370	0.0680	-1.8690
TC16	6.6910	0.0160	-6.6750
TC17	0.4370	0.4360	-0.0010
TC18	18.9760	0.3406	-18.6354
TC19	8.0000	0.0260	-7.9740
TC20	8.5440	1.1062	-7.4378
TC21	2.5640	0.0004	-2.5636
TC22	12.4070	1.7300	-10.6770
TC23	6.8700	2.5400	-4.3300
TC24	2.1240	0.0070	-2.1170
TC25	0.1890	3.2640	3.0750
TC26	8.6550	0.0560	-8.5990
TC27	7.3610	0.0110	-7.3500
TC28	0.8320	0.0004	-0.8316
TC29	7.8670	0.0240	-7.8430
TC30	8.6930	0.0120	-8.6810

the statistical techniques during the creation of the behavior maps, but the relatively small number of training cases used proved to not need the approach in training. However, its effectiveness alongside the contextually-weighted k-nearest neighbor algorithm becomes obvious in a side-by-side comparison of the chi-square statistics shown in Table 6-14. The table shows the chi-square statistic for each test case from the first draft and the final prototype. The last column shows the change in the statistic. A negative value in this column indicates that the win percentage of the prototype moved closer to the observed team's win percentage indicating that the addition of the statistical method to the k-nearest-neighbor algorithm did make a noticeable improvement in the performance of the prototype team.

Overall, this prototype showed that COLTS can effectively transfer both individual and collaborative behavior from an observed team to a new prototype team. Although, an excellent example, this prototype did not have particularly onerous the time constraints. The prototype described in Chapter 7 was chosen to stress COLTS with a much larger situation vector, larger number of potential situations, and a real-time time constraint.

CHAPTER 7 PROTOTYPE 3 - TEAMBOTS SMALL-SIZE ROBOTS

The final prototype developed for evaluating the effectiveness of COLTS used a more sophisticated action team than did the previous examples. The Teambots software [Balch, et al, 2000] was developed as a testbed for teams intending to compete in the RoboCup small-size robot competition. TeamBots provides a simulation of the small-size robot, a simulated ball, and a simulated soccer pitch. Researchers can test potential control systems for the robots within the simulation. The control system has access to the position information of all players and the ball as they would in the actual small-size robot competition. The control system has the ability to set the steer heading of the robots in radians, their speed and can also order them to kick at the ball. In addition to the basic simulation, several previously written team control systems are available to use as example teams and against which to compare. In this prototype, an existing team control system was chosen as the observed team from which to learn. This simulation and team have several characteristics that make the behavior of the team more difficult to imitate than those of the teams used in the previous prototypes. First, the simulation is continuous, not discrete. While the robot control software is called periodically as in the pursuit team, the simulated robots and ball are continuous move continuously between calls to the robot control software. Second, there are considerably more possible situations in this simulation. There are five players on each team plus one ball to monitor making a total number of 11 entities to track. Unlike the pursuit-evasion game, the positions are not discrete. The players can move anywhere within the soccer pitch, and the coordinates are floating point numbers rather than integers. This makes for an infinite number of possible positions unless the floating point numbers are truncated or rounded. In addition, the steering output is also a floating point number between $-\pi$

to π, which gives an infinite number of possible outputs unless the numbers are truncated or rounded. The other outputs are limited to a discrete number of possible outputs.

7.1 Observer Module

The observer module for the COLTS TeamBots prototype is described in the following sections. It was responsible for gathering data about the TeamBots team chosen for observation and formatting it as input to the learning module.

7.1.1 Observed Team and Simulation Environment

Fortunately, the TeamBots software has a similar structure to the previous games where the robot control system acts as an agent to the main simulation. Figure 7-1 shows a high-level block diagram of the TeamBots system. The SimulationCanvas is the controlling class of the simulation. It is responsible for calling the simulations of the physical robots and the ball as well as the robot control agents. The agents used for each team and type of robot simulation are driven by a configuration file input at start-up. The implementation of the observer module was structured similarly to the previous prototypes. Logging code functionality was added within the Simulation Canvas class before and after calls to the robot control software. A flag was added to allow this logging to be turned on or off at run-time. If the data are not needed for analysis of the game, it is desirable to turn the logging off as it does slow down the game and the log file size can easily grow to 20 to 30 MB.

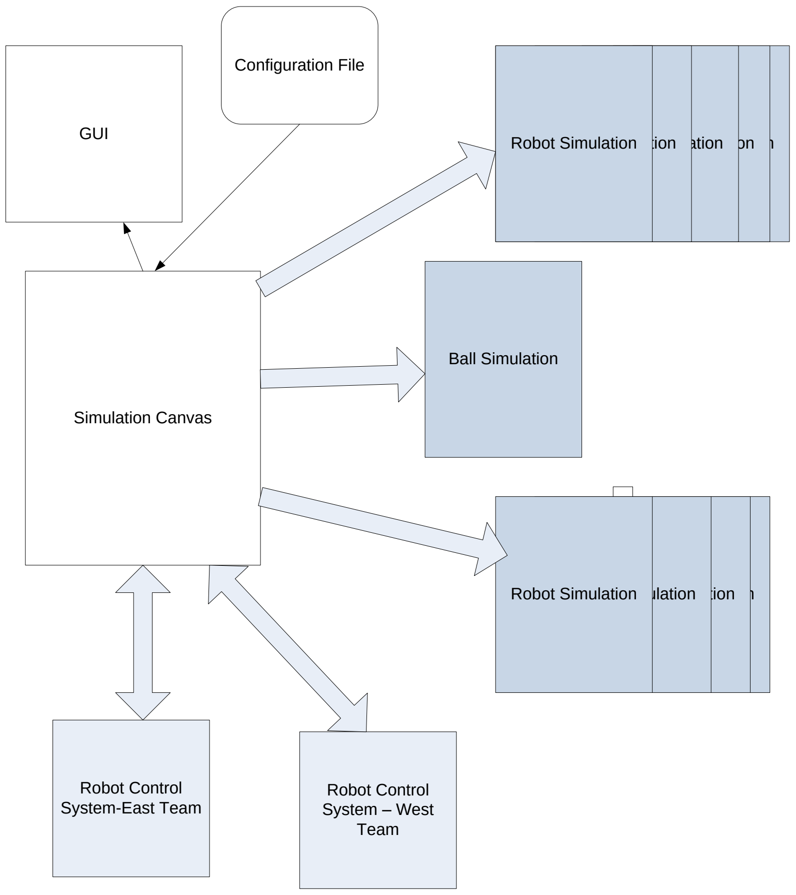


Figure 7-1 TeamBots Block Diagram

Positions logged at the end of the previous time-step will not be valid at beginning of the next time-step so the logging was done slightly differently from the previous prototypes. Prior to the calls to the robot control system software, all players' positions and the ball position are logged. Upon completion of the calls to the robot control system software, the observed team players' states, desired steering position and desired speed are likewise logged. This gives a complete picture of the positions at the beginning of the time-step, as well as the desired actions afterwards.

The next step was to choose a team to observe and imitate. In order to choose a team, a tournament among the teams provided with TeamBots was held. These teams were developed by various students throughout the years. Some are meant to be examples of how to build a robot control systems and others are tests of various strategies. Two teams were chosen. Initially, the AIKHomo team was chosen because of its successful win record in the tournament. Table 7-1 shows AIKHomo's win/lose record in the tournament. However, attempts to contextualize the AIKHomo team proved quite difficult. The state descriptions given in the log were meant to help the implementer debug the code, not give true descriptions of the roles taken. Since it was not possible to get usable contexts from the AIKHomo log, it was abandoned as the team to implement.

Table 7-1 AIKHomo Team Tournament Record

Opponent	**Op Wins**	**AIKHomo Wins**	**Ties**	**Op Total Pts**	**AIKHomo Total Pts**
SchemaDemo	0	10	0	8	47
CDTeamHetero	0	10	0	3	136
MattiHetero	4	5	1	24	27
DaveHeteroG	0	10	0	3	56
DoogHeteroG	0	10	0	14	55
FemmeBotsHeteroG	0	7	3	7	38
JunTeamHeteroG	0	9	1	2	221
Kechze	0	10	0	3	89
PermHomoG	0	10	0	3	47
Totals:	4	81	5	67	716

The SchemaDemo team was chosen next for its observable strategy and relatively few observable contexts. The SchemaDemo team's strategy was to have four of the five players attempting to get behind the ball and push it towards the goal while the fifth player serves as goalie. The contexts are relatively easy to see and determine from the logged data. The field players simply move toward the back of the ball. The "back" of the ball is the side opposite the goal. If the player is already behind the ball, they are in the GO_TO_BALL context and choose a steering heading towards the ball. If the player is not behind the ball, they are in GET_BEHIND_BALL context and attempt to move behind the ball without bumping it in a direction that could score for the opponents. The goalie has a single GOALIE context where the player moves vertically in front of the goal attempting to block the ball.

Table 7-2 SchemaDemo Team Tournament Record

Opponent	**Op Wins**	**SchemaDemo Wins**	**Ties**	**Op Total Pts**	**SchemaDemo Total Pts**
AIKHomoG	10	0	0	47	8
CDTeamHetero	1	7	2	16	37
MattiHetero	10	0	0	57	7
DaveHeteroG	5	2	3	23	19
DoogHeteroG	8	1	1	53	32
FemmeBotsHeteroG	6	2	2	226	18
JunTeamHeteroG	0	10	0	4	123
Kechze	4	5	1	30	36
PermHomoG	1	6	3	12	20
Totals:	45	33	12	468	300

SchemaDemo team did not have a particularly good win record in the tournament, but it was repeatable. Table 7-2 shows SchemaDemo's win/lose record against the other tournament team. A tournament with 10 games against each team was chosen because it produced repeatable results. Repeatability was demonstrated because a tournament with 100 games showed the same numbers multiplied by 10. This shows that the game results were highly repeatable. The key metrics from the tournament is number of goals scored by SchemaDemo and the number of goals scored against SchemaDemo. The win/lose/tie metrics are interesting, but do not give much input into true performance of the team because unless the teams are very mismatched games are typically very low scoring. One goal can make the difference between a win, a loss or a tie.

SchemaDemo was chosen for initial implementation. The log file chosen for input into the learning algorithm had AIKHomo as West Side team against SchemaDemo as East Side team. A single game was logged for use as observed, training data. Interestingly, the log of one game in TeamBots proved to be larger in size than the log files used in prototypes 1 and 2 combined. A sample of a single time period is shown in Appendix G. Table 7-3 describes the data available in the log at each time-step.

Table 7-3 Logged Data Types for each Time-step

Data Element	Sub Element	Type	Range
Simulation time		Integer milliseconds	
Player Information at start of time-step – Available for all 10 players		Complex – following SubElements	
	Side	Enumeration	East or West
	Number	Integer	0 to 4
	State Data	Descriptive String	
	Position x	Double	-1.37 to 1.37
	Position y	Double	-0.76 to 0.76
	Heading	Double	π to - π
	Speed	Double	0 to 1.0
Ball Position x		Double	-1.37 to 1.37
Ball Position y		Double	-0.76 to 0.76
Ball Heading		Double	π to - π
Player Information at end of time-step. East side only			
	Number	Integer	0 to 4
	State Data	Descriptive String	
	Heading	Double	π to - π
	Speed	Double	0 to 1.0
	Kicked	Boolean	True or False

7.1.2 Situation and Action Vector Specification

Next, the contents of the Situation class and possible Actions were determined. Table 7-4 shows the data elements used to form the Situation class. All of the information is available from or is computable from the information in the log file and from the sensors provided to each team member at run-time. Note that not all the information available in the log file was available at run-time. For example, the positions of the ball, teammates, and opponents can be determined at run-time. However, their heading and speeds are not available at run-time. Therefore, these data were not used as part of the situation vector. Table 7-5 list the elements of the action vector. In this simulation, the action vector a data structure consisting of three data elements. These elements correspond to the commands given to the robot in each time-step of the simulation.

Table 7-4 TeamBots Prototype Situation Vector Data Elements

Element Name	**Description**	**Data Type**
myPosition	x, y coordinates of this team member	double, double
playerNum	The player number assigned to this team member	integer
teammates[4]	An array containing the x,y coordinates of each team member	Array of 4 double, double
opponents[5]	An array containing the x,y coordinates of each opponent team member	Array of 5 double, double
ballPosition	The x,y position of the ball	double, double
ballAngle	A computed angle from this team member to the ball.	double
isBehindBall	A computed true/false indicating whether team member is currently behind ball.	boolean

Table 7-5 Action Class Data Elements

Element Name	**Description**	**Data Type**
steerHeading	Heading of the robot. Floating point value between 0 and 2π. Angle of 0 is due east.	double
speed	Speed of the robot. Floating point value between 0 and 1.0 where 1.0 is the full speed of the robot and 0 is full stop.	double
kick	Request for robot to kick.	boolean

7.1.3 Contextualization

Before modifying the Trainer class for the TeamBots log file, identification of contexts in the observed team was required. SchemaDemo was chosen because of the straight-forwardness of the observed team's behavior. Players 1 to 4 attempt to place themselves in a position behind the ball moving towards the opposite goal. Player 0 stays in the goal box protecting the goal unless the ball approaches. When a ball approaches, player 0 will move away from goal to kick ball back away from the protected goal. Players 1 to 4, which are the offensive players have two basic contexts.

- If they are already in position behind the goal, they continue to move to the ball at an angle that will take both into the opposition goal in order to score.
- If the ball is in front of the player, they attempt to move into position behind the ball without accidentally striking it in the wrong direction.

This results in two contexts for the offensive player, MOVE_TO_BALL and GET_BEHIND_BALL. Player 0 or the goalie has a single additional context, MOVE_TO_BACKFIELD. These three contexts were used for the observed parsing of the log file. The Goalie will also use the GET_BEHIND_BALL context when the ball is very close to the goal. Figure 7-2 and Figure 7-3 show the contexts initially chosen for the prototype.

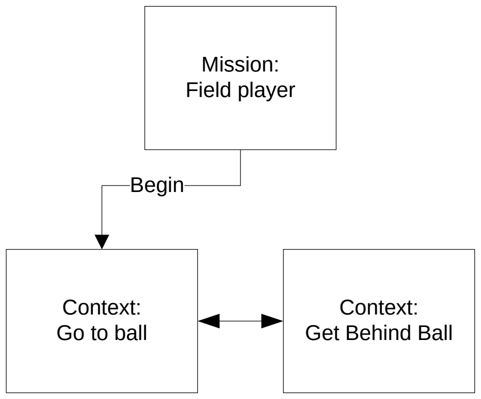


Figure 7-2 Context for Field Player

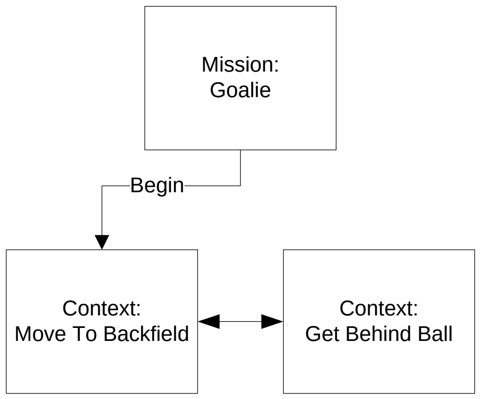
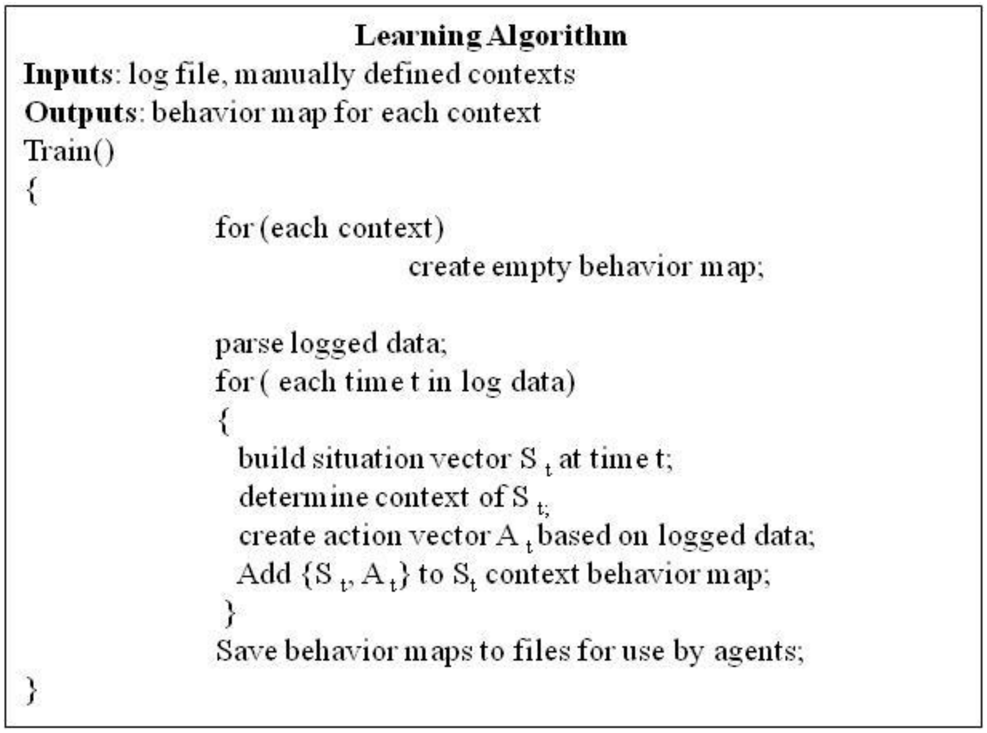


Figure 7-3 Contexts for Goalie

7.2 Learner Module

The algorithm of the Trainer class had to be modified slightly because of the continuous nature of TeamBots. The continuous movement meant that the context and positions in S_{t-1} were not necessarily related in any way to $S_{t,}$ as they were in the two previous prototypes. The observer module supported the change by showing initial positions in each time-step followed by resulting actions. Figure 7-4 shows the modified training algorithm. It was actually easier to implement than the either or the previous prototypes as the initial S for each time-step is directly associated with the actions to take. For field players, context was determined by the position of the player being analyzed versus the position of the ball. If the ball was between the player and the desired goal, the context was MOVE_TO_BALL. If the ball was between the player and its own goal, the context was GET_BEHIND_BALL. For the goalie, context defaulted to MOVE_TO_BACKFIELD unless the ball was between the player and the defended goal. In that case, the context became GET_BEHIND_BALL.

Unfortunately, the behavior maps produced by the initial run of the Trainer class were so large that when used by the run-time agents, the simulation ran painfully slow. In an attempt to alleviate this issue, additional contexts were added. The new contexts take the two initial contexts of MOVE_TO_BALL and GET_BEHIND_BALL and create two contexts for each player number. The resulting contexts are listed in Figure 7-5. The behavior maps generated with these contexts were used as input to the initial run-time agents. To determine the context, player number was added to the observed criteria.



Learning Algorithm

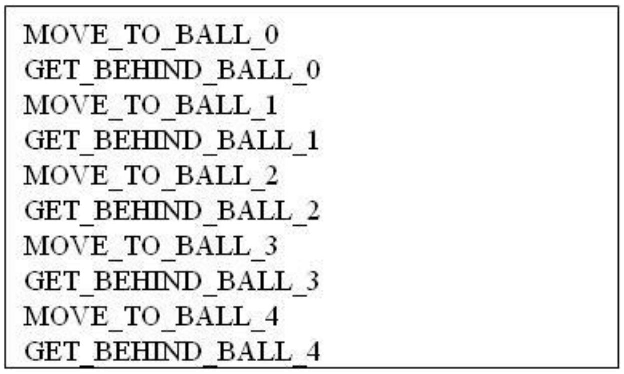
Inputs: log file, manually defined contexts
Outputs: behavior map for each context

```
Train()
{
        for (each context)
                create empty behavior map;

        parse logged data;
        for ( each time t in log data)
        {
          build situation vector $S_t$ at time t;
          determine context of $S_t$;
          create action vector $A_t$ based on logged data;
          Add {$S_t$, $A_t$} to $S_t$ context behavior map;
        }
        Save behavior maps to files for use by agents;
}
```

Figure 7-4 Training Algorithm for TeamBots Prototype

The observed team required less than 2 ms to process all five players. The two milliseconds actually included both teams and the simulation of the ball and robot hardware. Even when the additional contexts were added, this time was still an average of 54 milliseconds indicating that COLTS required nearly 52 ms to run. This increased execution time is a direct result of the size of the behavior maps. It is also far above the goal of less than 20 milliseconds.



```
MOVE_TO_BALL_0
GET_BEHIND_BALL_0
MOVE_TO_BALL_1
GET_BEHIND_BALL_1
MOVE_TO_BALL_2
GET_BEHIND_BALL_2
MOVE_TO_BALL_3
GET_BEHIND_BALL_3
MOVE_TO_BALL_4
GET_BEHIND_BALL_4
```

Figure 7-5 Final SchemaDemo Contexts

The Trainer class always prevents exact duplicate situations from being added to the behavior maps to prevent excessive comparisons at run-time. However, because so many of the data elements in the Situation class are floating point types, it is highly unlikely that exact duplicates exist. It is much more likely that Situation classes are differentiated by very small decimal points. So the training algorithm was redone to not just eliminate exact duplicates, but to also eliminate very similar duplicates. Similar duplicates are determined by running the k-nearest neighbor algorithm with weights of 1.0 for all elements. The results are compared against a constant value called SIMILAR. If the value returned by the k-nearest neighbor algorithm is less than SIMILAR, the two situations are deemed similar and the new similar Situation is not added to the behavior map. Behavior maps were developed for several different values of SIMILAR.

Action vectors were created by parsing the log file for the actions taken at the end of the time-step. The values read include heading, speed, and whether or not to kick. Because the actions were paired with the observed situation data in the log file, the need to compare to situation vectors at adjacent times was eliminated in this prototype.

7.3 <u>Run-Time Module</u>

The run-time modules consist of the agent implementation and the contextually-weighted k-nearest neighbor implementation. These are described in the following sections.

7.3.1 Agent Implementation

While the basic structure of the COLTS run-time agents from the previous prototypes was an excellent starting point, it was necessary to make some changes to the COLTS agents for TeamBots. Because of the continuous vs. discrete nature of TeamBots, it cannot be assumed that the context from the previous time-step is still the valid context. Like the other prototypes, the first step in the agent is to build its current Situation vector based on information from sensors. A necessary additional step is to determine the current context. A variety of techniques were considered, including using a type of behavior map that returns a context rather than a behavior. However, since context is primarily determined by two variables, player number and angle to ball, a decision tree was used to determine context. Although not automated, this is the fastest-executing way to determine context. Execution time is a very strong constraint in this simulation so automation was sacrificed for speed.

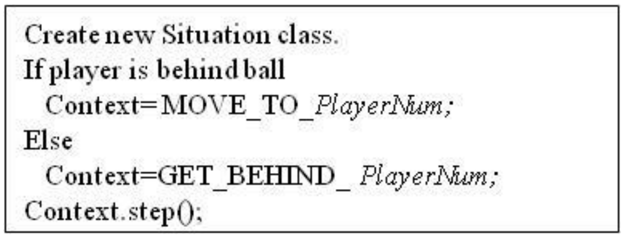

```
Create new Situation class.
If player is behind ball
  Context=MOVE_TO_PlayerNum;
Else
  Context=GET_BEHIND_ PlayerNum;
Context.step();
```

Figure 7-6 TeamBots CCxBR Agent Algorithm

Finally, the agent determines its appropriate action in its current context using the behavior map for that context. Figure 7-6 shows the algorithm of the decision tree for the COLTS agent used to determine context. Figure 7-7 shows a UML class diagram of the COLTS agents designed to play as a team in TeamBots. The class diagram shows a lot of similarities to the previous prototypes. The primary differences are in the names of the contexts instantiated by the TeamContext class and of course, the data used in the Situation class representing the state vector. Figure 7-8 shows where this team fits into the TeamBots simulation block diagram. The COLTS run-time agents replace the East Team originally represented by SchemaDemo.

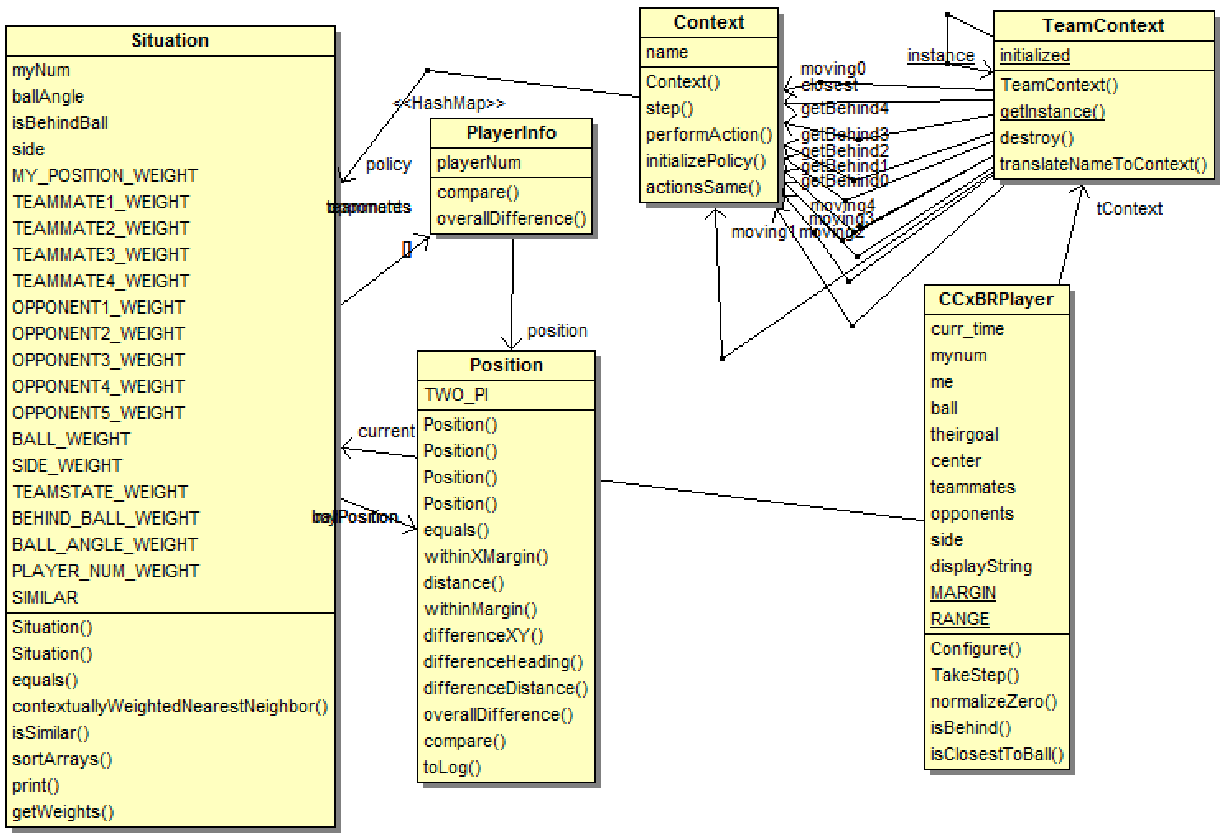


Figure 7-7 TeamBots Prototype Run-time Agent UML

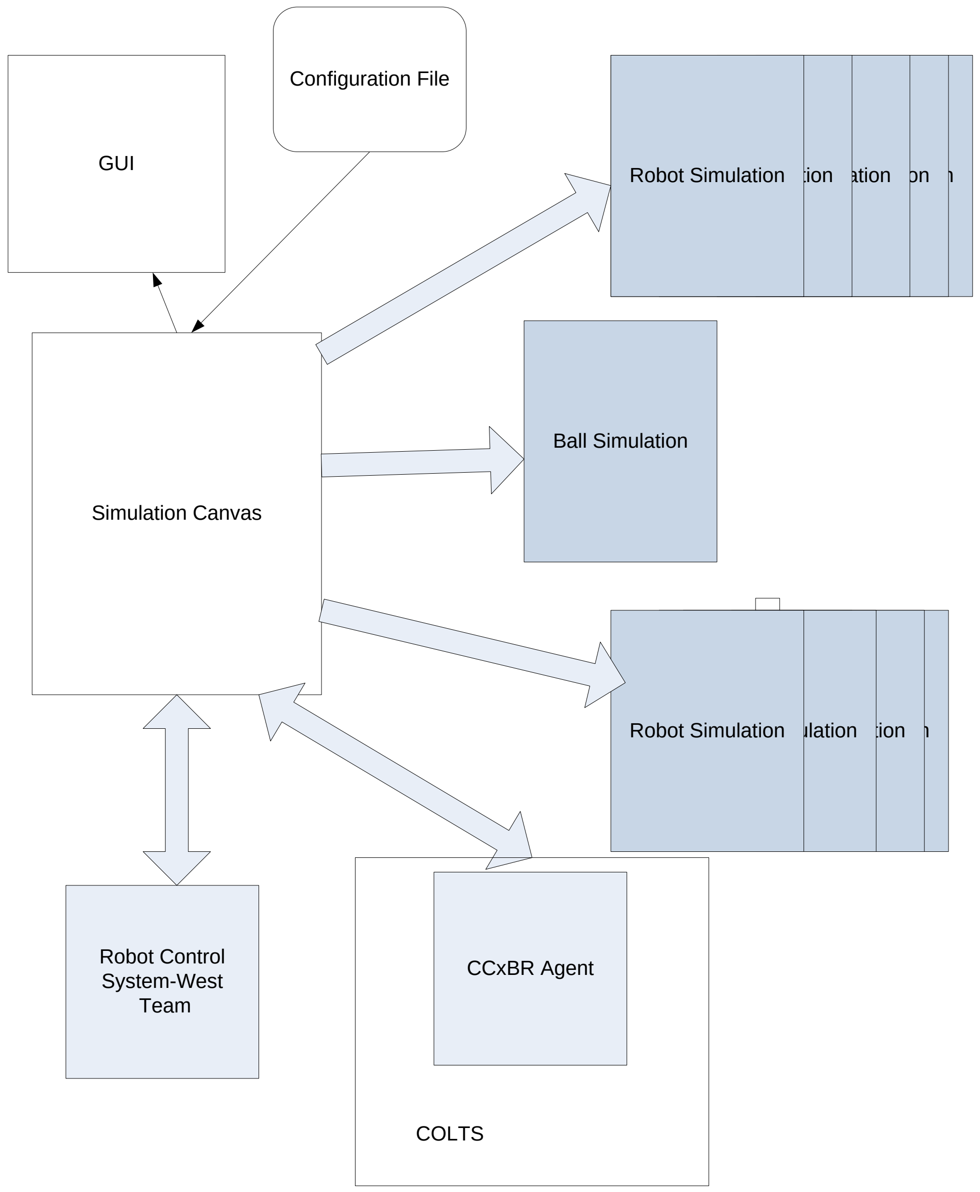


Figure 7-8 TeamBots Block Diagram with COLTS team

7.3.2 Contextually-weighted k-Nearest Neighbor Algorithm

Once the behavior maps were populated by the training algorithm and the run-time agents were in place, the next step was to develop the weights for the contextually weighted k-nearest-neighbor algorithm. Although there are a total of ten possible contexts for the agents, the contextual weighting is divided into three categories.

1. The first category is the goalie category and includes only the context MOVING_0. The behavior of this particular player was different enough to warrant its own context.
2. The second category is the moving category. This includes the MOVING_1, MOVING_2, MOVING_3 AND MOVING_4 contexts. The behavior of these players when behind the ball is similar enough to warrant the same weights.
3. The third and final category is the get_behind category. This includes all the GET_BEHIND_# contexts. Like the moving category, the behavior of all the players when trying to get behind the ball is similar enough to warrant identical weights.

As with the previous prototypes, the weight vector was determined through trial-and-error experimentation. In this case, the experimentation criteria used was a game against the AIKHomoG team. This team was chosen as the opponent team because it was the opponent in the training data. The final weights shown in Table 7-6 had the best performance against this team in the final prototype.

Table 7-6 Final Contextually-based Weights

Data Element Name	**MOVING_0 weight**	**MOVING weight**	**GET_BEHIND weight**
MY_POSITION_WEIGHT	1.0	1.0	1.0
TEAMMATE1_WEIGHT	0.0	0.1	0.1
TEAMMATE2_WEIGHT	0.0	0.1	0.1
TEAMMATE3_WEIGHT	0.0	0.1	0.1
TEAMMATE4_WEIGHT	0.0	0.1	0.1
OPPONENT1_WEIGHT	0.0	0.0	0.0
OPPONENT2_WEIGHT	0.0	0.0	0.0
OPPONENT3_WEIGHT	0.0	0.0	0.0
OPPONENT4_WEIGHT	0.0	0.0	0.0
OPPONENT5_WEIGHT	0.0	0.0	0.0
SIDE_WEIGHT	0.0	0.0	0.0
BALL_WEIGHT	1.0	0.5	1.0
TEAMSTATE_WEIGHT	0.0	0.0	0.0
BEHIND_BALL_WEIGHT	0.0	1.0	0.0
BALL_ANGLE_WEIGHT	2.0	1.0	2.0
PLAYER_NUM_WEIGHT	1.0	1.0	1.0

7.4 Results and Conclusions

The following sections describe the criteria for evaluating the COLTS TeamBots team and the results of the experimentation. In addition, some changes to the initial team made to improve performance and execution time are described.

7.4.1 Criteria for Evaluation

In order to evaluate the success of the COLTS team, a tournament was run against the same opponents that SchemaDemo played when evaluating which team to observe. Table 7-2 shows SchemaDemo's performance against the various opponents. The two most important data elements in this evaluation are total points scored by SchemaDemo and total points scored by opponents. The first shows the effectiveness of the transfer of offensive behavior and the second the effectiveness of the defensive behavior. Since this prototype is designed to stress the COLTS system, some criteria for partial success was also needed. To serve as such criteria, two additional tournaments were held. One replaced the COLTS team with a team that simply remained in their initial positions. The second replaced the COLTS team with a team of players that moved randomly. The results of these tournaments are shown in Table 7-7 and Table 7-8. The points scored by the non-moving team were actually the result of the opponent team playing particularly badly. The random team saw the opponents score even more points than the non-moving team. This is probably based upon the fact that one of the non-moving team members was always in goalie position which prevented some goals simply by serving as an obstacle.

In addition to the performance criteria, the COLTS TeamBots team had a limited time in which to execute. Although the total time between execution of each agent is 50 ms, this time allotment must be divided between both teams and the time to run the ball simulation and update the GUI. Fortunately, the current execution of the simulation elements on an Intel Core i7 2.67

GHz processor running Windows Vista takes less than 1 ms, but it seemed reasonable to allot 20% of the available time to the simulation itself for a total of 10 ms so that the simulation can run on slower processors. That leaves 40 ms to divide evenly between the two teams for a total of 20ms each.

Table 7-7 Tournament Results for Non-moving Team

Opponent	Opponent Wins	Non-Moving Wins	Ties	Opponent Total Points	Non-Moving Total Points
AIKHomoG	10	0	0	356	0
CDTeamHetero	10	0	0	165	0
MattiHetero	10	0	0	197	2
DaveHeteroG	10	0	0	18	0
DoogHeteroG	10	0	0	90	0
FemmeBotsHeteroG	10	0	0	10	0
JunTeamHeteroG	10	0	0	293	1
Kechze	10	0	0	10	0
PermHomoG	9	0	1	10	0
Totals:				**1149**	**3**

Table 7-8 Tournament Results for Random Team

Opponent	Opponent Wins	Random Wins	Ties	Opponent Total Pts	Random Total Pts
AIKHomoG	10	0	0	361	0
CDTeamHetero	10	0	0	234	0
MattiHetero	10	0	0	405	0
DaveHeteroG	10	0	0	90	0
DoogHeteroG	10	0	0	280	0
FemmeBotsHeteroG	10	0	0	15	0
JunTeamHeteroG	10	0	0	271	2
Kechze	10	0	0	525	0
PermHomoG	10	0	0	125	0
Totals:				**2306**	**2**

7.4.2 Initial Results

The experimental results of the COLTS TeamBots prototype showed some partial success. However, it became clear that this approach is not well suited to the TeamBots simulation. The initial COLTS team was totally unrecognizable as the observed team. The first team used behavior maps generated with a SIMILAR value of 0.0. The results were so bad that it did not seem worthwhile to even run a tournament to see if the team performed better than the randomly-moving or stationary teams.

7.4.3 Change in Approach

7.4.3.1 Background

This initial failure prompted a more in-depth review of Floyd, et al [2008] who successfully used a similar method for a single soccer player. The review made it clear that Floyd, et al had limited the number of actions to a set of primitive behaviors. This limitation is also a significant element of Bentivegna's [2004] research. Rather than attempting to tell his robotic arm which specific motors to turn and how much, Bentivegna created a set of primitive behaviors that were translated to specific angles and forces. Floyd, et al [2008] did the same by creating a set of player behaviors such as dash or move towards ball, rather than attempting to learn specific angles and speeds for the player. Since the research by both investigators was the inspiration for COLTS learning algorithm and behavior maps, a set of core behaviors was developed that reflected the behaviors of the observed SchemaDemo team.

7.4.3.2 New Primitive Behaviors

The new primitive COLTS TeamBots behaviors were developed based upon the state descriptions given to each player by the developer of the SchemaDemo code. This description was part of the data in the log file. In total, only three behaviors were needed to emulate the

SchemaDemo team. These were MOVE_TO_BALL, GET_BEHIND_BALL, and MOVE_TO_BACKFIELD. Code was written to implement each of these behaviors.

- For MOVE_TO_BALL, the player moved toward the ball. The steer heading was provided by a vector class named Vec2 that calculated the angle between the player and ball based on their current positions. If the player was close enough the ball, it would kick towards the opposite goal.
- The GET_BEHIND_BALL behavior chose a steer heading to place the player between the ball and their own defended goal. If the ball was directly on that vector, the player would move around the ball.
- The most complex of the primitive behaviors was MOVE_TO_BACKFIELD. This behavior was basically goalie behavior. The player so instructed would move to a point slightly in front of and in the center of the defended goal. If already in the defense position, the player would remain stationary.

This change required that the behavior maps be retrained to accept an enumeration describing one of the three behaviors rather than the Action class developed to hold steer heading and speed. The change was relatively easy and the behavior maps were retrained using the same original log data.

7.4.3.3 Primitive Behavior Results

The resulting behavior was visually quite similar to the observed SchemaDemo team. However, the processing time for the COLTS prototype was significantly longer than the observed team. Table 7-9 show the tournament results for the too-slow COLTS team. These results show that the COLTS team scores significantly more goals than the non-moving and random teams and the opposing teams score significantly less goals. This indicates that the COLTS team did learn some soccer playing skills. However, Table 7-10 shows the differences between the tournament results of the observed team and the COLTS prototype. Statistical analysis is not needed to show that the COLTS prototype did not effectively duplicate the SchemaDemo team. The COLTS prototype did not win any games and while it did better than the non-moving team, the results did not approach that of the observed team.

Table 7-9 Tournament Results for COLTS

Opponent	**Op Wins**	**COLTS Wins**	**Ties**	**Op Total**	**COLTS Total**
AIKHomoG	10	0	0	159	0
CDTeamHetero	7	0	3	37	14
MattiHetero	6	0	4	18	1
DaveHeteroG	7	0	3	16	3
DoogHeteroG	10	0	0	147	8
FemmeBotsHeteroG	9	0	1	43	1
JunTeamHeteroG	8	0	2	69	32
Kechze	10	0	0	80	10
PermHomoG	2	0	8	3	0
Totals:				**572**	**69**

Table 7-10 Tournament Difference between SchemaDemo and COLTS

Opponent	**Op Wins**	**COLTS Wins**	**Ties**	**Op Total**	**COLTS Total**
AIKHomoG	0	0	0	112	-8
CDTeamHetero	6	-7	1	21	-23
MattiHetero	-4	0	4	-39	-6
DaveHeteroG	2	-2	0	-7	-16
DoogHeteroG	2	-1	-1	94	-24
FemmeBotsHeteroG	3	-1	-2	24	-9
JunTeamHeteroG	8	-10	2	65	-91
Kechze	6	-5	-1	50	-26
PermHomoG	1	-6	5	-9	-20
Totals:				**311**	**-223**

Without additional training data, it is not possible to improve the scoring and defensive playing of the COLTS prototype team. However, it is possible to improve the playing time. Although the algorithm is optimized to run as quickly as possible, a very large behavior map will take significant time to compute. Therefore, once performance was optimized as much as possible with the available training data, the various sets of behavior maps developed with different values of SIMILAR were tested.

7.4.4 Execution Time Improvement

The difference in execution time based on the value of SIMILAR can be seen in Figure 7-9. These data clearly show that there are quite similar Situation values in the original behavior maps. The time to execute decreases sharply with a similar value of only 0.5 and continues to shrink until the difference between the values of 1.5 and 2.0 is indistinguishable. This does not, however, address whether or not this shrinking of the behavior maps impacts performance.

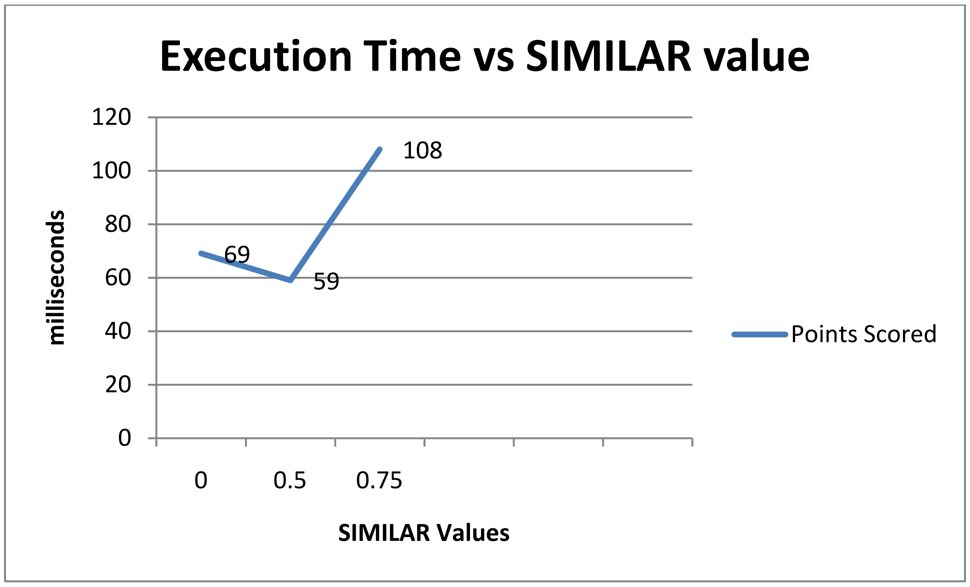


Figure 7-9 Impact of Training Changes on Execution Time

The execution time at a SIMILAR value of 0.5 is nearly within the acceptable range at a value of 27 ms. The execution times with SIMILAR values at 0.75 and above are clearly within the desired range. In order to determine if the performance was impacted by the smaller behavior maps, additional tournaments were performed with teams trained with the various SIMILAR values. The most indicative results are those against the AIKHomoG team, since that is the team trained against. The prototype trained with a SIMILAR value of 0.75 performed even better against AIKHomoG than the prototype trained with SIMILAR value of 0.0 and 0.5. The performance at higher values of SIMILAR began to decline.

Performance was determined by the net difference in goals scored and goals scored against the prototype team. The observed SchemaDemo team scored a total of 8 goals against AIKHomoG and gave up 47 goals. The too-slow COLTS prototype trained with SIMILAR value = 0 scored no goals and gave up 159 goals. The COLTS prototype trained with SIMILAR value = 0.75 scored no goals, but only gave up 143 goals. The change is small but significant.

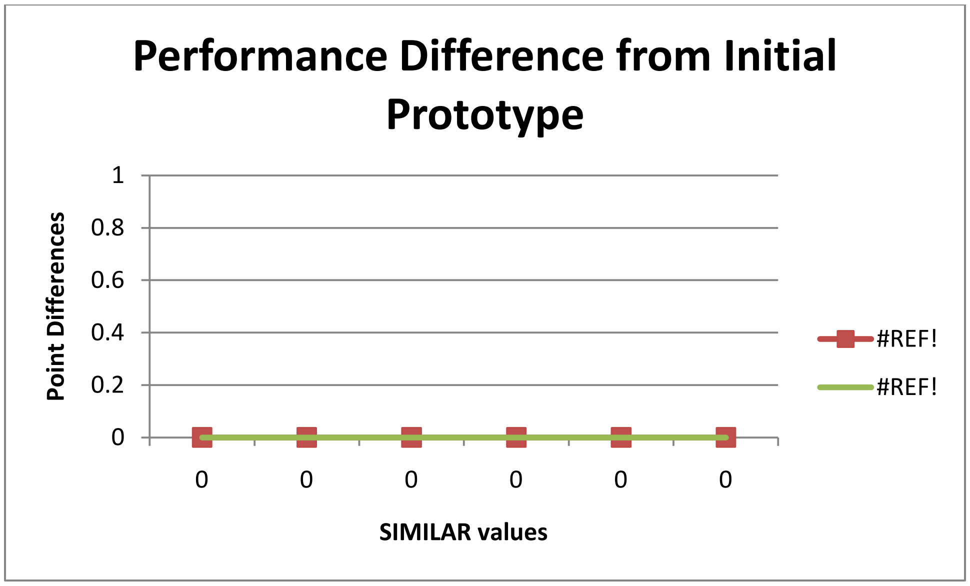


Figure 7-10 Performance Changes

Figure 7-10 shows how the performance against AIKHomo changed with the different values of SIMILAR. The performance is defined as the sums of the difference in points scored and points given away. The better performance at values of SIMILAR at 0.75 and 1.0 is seen in this graph.

7.4.5 Final Results

A value of SIMILAR = 0.75 was chosen as the final value for use. The final tournament results of the prototype trained with SIMILAR = 0.75 is shown in Table 7-11. Although a SIMILAR value of 1.0 performed well with AIKHomoG, values of SIMILAR above 0.75 showed a decrease in performance with the other teams. So, SIMILAR = 0.75 was chosen as the optimal value for both execution time and performance.

Table 7-11 Tournament Results for Prototype with SIMILAR = 0.75

Opponent	Op Wins	COLTS Wins	Ties	Op Total	COLTS Total
AIKHomoG	10	0	0	143	0
CDTeamHetero	10	0	0	79	28
MattiHetero	9	0	1	24	1
DaveHeteroG	7	0	3	17	2
DoogHeteroG	10	0	0	152	6
FemmeBotsHeteroG	10	0	0	66	2
JunTeamHeteroG	5	2	3	76	54
Kechze	9	1	0	126	15
PermHomoG	6	0	4	9	0
Totals:				**692**	**108**

7.5 Conclusions

The results of the COLTS TeamBots prototype show that the COLTS learning algorithm and run-time behavior function have limitations. The large number of possible situations and actions in the game makes it difficult to create behavior maps of a small enough size to run within the time limits of the game. The execution time is linked to the size of the situation vector and the size of the behavior map. In order to find the best match within the behavior map, each situation vector stored in the map is compared to the current situation using the contextually-weighted k-nearest-neighbor algorithm. The size of the situation vector determines how many calculations are done for each entry, and the size of the behavior maps determine how many times the algorithm is run. The results also show that a too small behavior map results in suboptimal performance of the prototype team. It was clear that it is not possible to create a large enough behavior map to accurately recreate the behavior of the observed team in this domain. The behavior maps only partially recreated the behavior of the observed team. The only way to get a team with better skills would be to include training data from several games against different opponents, as was done in the Pursuit-Evasion prototype. That training data included a total of 150 games with 15 different starting positions. However, the size of the training data for a TeamBots game is significantly larger and it would be unlikely that the size of the behavior map would be small enough to allow execution of the contextually-weighted k-nearest-neighbor algorithm within the allotted execution time.

Another issue with the TeamBots prototype was the difficulty in identifying contexts of the various teams. As the tournament shows, a variety of teams were available to test against, but very few of them had contexts or even repeatable primitive behaviors that were obvious simply by observing the players visually or through the log files. It is possible that the use of an expert during the contextualization would have enabled the development of usable contexts for the AIKHomo team which was more sophisticated than SchemaDemo. SchemaDemo, the team finally selected for imitation, was easily contextualized, but that was because of the simplicity of the implementation. Contextualization of this team was almost a break-down into the primitive behaviors. So while there was some success in duplicating the behaviors of the SchemaDemo team, the implementation took significantly longer to execute and required a great deal more memory than the observed implementation. So while it is possible to duplicate behavior to some extent using this technique on TeamBots and similar simulations, it simply does not make sense to do so.

CHAPTER 8 CONCLUSIONS AND FUTURE WORK

This chapter gives a brief summary of the research presented in this dissertation, followed by conclusions about the work. The final section offers recommendations for future uses of COLTS, and discusses possibilities for future studies.

8.1 Summary

This dissertation presented a semi-automated method of learning collaborative behavior via observation. The hypothesis is that the combination of a multi-agent framework designed to promote effective teamwork with a proven single-entity learning-by-observation method would prove successful in learning effective collaborative behavior. Before beginning the experiment, it was necessary to choose a multi-agent framework and one or more learning-by-observation methods to combine into COLTS.

Psychological studies about the nature of teamwork and what makes teams effective were reviewed to lay a solid foundation for this work. This study showed that most effective teams, regardless of type or task, had a shared mental model. The shared mental model of an effective team included information about the goals of the team and the role of each team member. This psychological model is reflected in Joint Intention Theory [Cohen & Levesque, 1991]. JIT is a group of definitions and theorems that represent one of the dominant theories on modeling collaborative behavior. One of the main ideas of JIT is that all members of a team must share a common view of the goals of the team. So in order to effectively learn collaborative behavior, the agent framework used must be capable of accurately representing collaborative behavior by implementing a theory such as JIT. Members of an effective team must also be capable of performing and understanding their individual tasks and goals in addition to the commonly shared goals. So an agent framework was needed to effectively provide elements of JIT and be able to accurately model individual behavior as well. Collaborative Context-based Reasoning (CCxBR) was chosen as the paradigm used for the multi-agent framework. CCxBR has been shown to have the ability to successfully implement effective and efficient modeling of collaborative behavior. [Barrett, 2007] It is also an extension of the single-agent paradigm Context-based Reasoning. [Gonzalez, et al, 2008] Context-based Reasoning has been used to effectively model and simulate human behavior in a number of applications.

Once CCxBR was chosen as the framework for the multi-agent team, it was necessary to develop a learning-by-observation technique that is able to scale from a single-entity to a team of entities. Ideally, the technique would scale linearly in terms of memory use and processing time. A variety of techniques in learning-by-observation and learning-by-demonstration were reviewed. The ultimate choice for a learning-by-observation method was inspired by case-based reasoning techniques [Floyd, et al, 2008] and robotic learning-by-demonstration memory-based policies. [Bentivegna, 2004] Behavior maps perform the same function in COLTS as Bentivegna's [2004] memory-based policy and Floyd, et al's [2008] case-base. If used as is, this technique would result in a single behavior map being developed for each team member. However, since CCxBR breaks the problem into contexts, the decision was made to create a behavior map for each context. Because some contexts are shared by multiple agents, the number of behavior maps created had the potential to be less than the number of behavior maps used by one agent multiplied by the number of team members.

In order to test the effectiveness and adaptability of COLTS, three separate prototypes were developed. The first prototype learned the behavior of a simulated bucket brigade. While not a particularly difficult task to simulate because of limited behaviors and few possible situations, the bucket brigade provided an excellent test bed for developing an implementation of

CCxBR and the learning algorithm in Java. The COLTS prototype bucket brigade was able to successfully duplicate and generalize the behavior of the observed team, including collaborative efforts such as handing a bucket from one member to another.

The second prototype of a pursuit-evasion game actually proved that COLTS is capable of effectively imitating the individual and collaborative behavior of a team. COLTS was able to effectively imitate the behavior of the four-man pursuit team in situations matching those trained. The prototype statistically matched the win ratio of the 15 trained situations and 15 non-trained situations. Matching was defined as being within the 95% confidence interval when a chi-square statistical test was performed between the observed and prototype win ratios. In addition, the prototype was able to generalize the behavior of the team as well in a series of tests against a smarter evader. When the same 30 situations were run against the smarter evader, only 28 of the 30 were able to pass the chi-square test within the 95% confidence interval. However, this still represents an excellent generalization of the behavior learned.

The third prototype, on the other hand, showed that COLTS does have limits and is not an appropriate method for some types of simulations with a very large number of possible situations and actions. The TeamBots simulation of RoboCup's small-size robot league was used as the basis of this prototype. A previously developed team was used as the observed team. This simulation differed greatly from the ones used in the previous prototypes in a number of ways. First, this simulation was continuous. The movement of the robots and the ball was modeled even when the control system was not called

The five-man teams also represented a much larger set of state data to track in each time-step as well. In addition to a larger set of state data, the number of possible values of each piece of data was much larger as well. The time allowed to run each time-step was also significantly smaller than the previous simulations because of the continuous nature of the simulation. This prototype was intended to stress the COLTS algorithm and find the algorithm's weaknesses. It certainly succeeded in that task. Some behavior was duplicated, but the COLTS prototype team was not able to accurately duplicate the observed team's behavior or meet the time constraints. This does not represent a reason to dismiss COLTS. Argall, et al [2009] noted in their survey of learning by demonstration that continuously modeled systems typically used a different approach to learning than discretized systems. The failure of prototype 3 does indicate that the algorithm has limitations in the continuous domain, and additional work is needed to expand it for use on such applications.

8.2 Conclusions

The experiments showed once again that the CCxBR paradigm is effective as a teamwork framework and provided an excellent implementation of CCxBR in the Java language. This framework could quite easily be adapted for use with a different type of behavior function. The behavior maps were the cause of the timing issues of the TeamBots prototype, not the agent framework. However, better contextualization of the observed team could also reduce the amount of time needed to process the behavior map as well. CxBR, the basis of CCxBR, has been shown to work well with a variety of different types of behavior functions [Fernlund, 2004] [Fernlund, et al, 2009] [Gonzalez, et al, 2002] [Gonzalez, et al, 2008] [Stensrud & Gonzalez, 2008] so CCxBR should have the same capability. To date, Barrett [2007] has shown it works with hard-coded functions and COLTS has shown it to work with behavior maps.

Behavior maps have been shown to be an effective and efficient behavior function in discrete time-stepped simulations with a limited number of possible situations such as the bucket brigade. The pursuit-evasion game showed that it can be effectively used when a much larger number of possible situations exist when contextualization is properly used to break the task down. However, TeamBots showed that without appropriate contextualization, an infinite number of possible situations will make behavior maps virtually unusable.

The main contribution of this dissertation is a novel approach to learning collaborative behavior via observation. In conjunction with CCxBR agents, behavior maps proved an effective technique for capturing behavior from one team to another. Although only simulated teams were used as input in this dissertation, the approach could work just as well for learning collaborative human behavior, provided that the observer module was modified to capture human behavior as well. In addition, a new, generic implementation of the CCxBR framework is provided as part of this approach. The source code used for each of the prototypes is provided in the appendices of this dissertation. While each prototype is presented separately, inspection of the run-time portion reveals a great deal of code reuse from prototype to prototype. This code could quite easily be adapted for another COLTS prototype or used with another type of behavior functions.

8.3 Future Work

The most obvious need for additional work revealed by this dissertation is the need for an effective algorithm to automatically provide contexts both for the individual team members and the teams. Several possible approaches to the problem are worth pursuing. The first possible approach would be the expansion of the work done by Trinh [2009] in automatically determining context for a single-entity into teamwork. The second possible approach would be the expansion of work into teamwork pattern recognition. [Luotsinen, et al, 2007] [Sukthankar, et al, 2006] There is, of course, always the possibility of a hybrid of the two methods.

In addition to contextualization, the other area done manually in this investigation that could be automated would be the determination of weights for the contextually-weighted k-nearest-neighbor algorithm. There has been some work into using genetic algorithms to automatically determine weights in a k-nearest-neighbor algorithm. [Floyd, et al, 2008] This work could be expanded for use into the algorithm used by COLTS providing that a fitness function could be developed that would test only for a particular context.

Since the learning algorithm of COLTS was unable to effectively reproduce the behavior in the TeamBots, another avenue for future work would be to use an on-line learning algorithm to improve a behavior map after the initial off-line observation and learning. A reinforcement algorithm able to improve the entries in the behavior map so that more effective behaviors were stored could improve the behavior of the team significantly. This would be comparable to the work done with FALCONET [Stein, 2009] where a task was initially learned from observation and the performance improved upon using experiential and instruction learning techniques. This type of learning could be used even in cases where behavior was adequately learned to make the team even better. For example, the pursuit-evasion team imitated in prototype 2 was most ineffective when the starting positions of the pursuers were far away from the initial position of the evader. A reinforcement algorithm could be used to develop a more efficient search algorithm for the pursuit team than the observed team had.

APPENDIX A – BUCKET BRIGADE TRAINER CLASS

Trainer.java

```
package Training;
// main class for training algorithm for bucket brigade
import Common.Constants;
import Common.Situation;


import java.io.BufferedReader;
import java.io.EOFException;
import java.io.FileReader;
import java.io.FileOutputStream;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;

import java.io.ObjectOutputStream;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Iterator;
import java.util.List;
import java.util.Map;
import java.util.Set;
import java.util.Vector;


public class Trainer implements Constants
{
	private BufferedReader in;
	private String logName = null;
	private List<Map<Situation, Actions[]>> maps = new ArrayList<Map<Situation,
Actions[]>>();
	private Vector[] trainingData = new Vector[11];


	public Trainer(String fileName)
	{

	   logName = fileName;
		try
		{
		// open file for reading
			in= new BufferedReader(new FileReader(logName));
		}
		catch (Exception ex)
		{
			System.out.println("Error opening log file");
			ex.printStackTrace();
		}

		//create a hash map for each context
		for (int i=0; i<= 10; i++)
		{
			maps.add(new HashMap<Situation, Actions[]>());
			trainingData[i] = new Vector();
		}

	}

	public void createHashMaps()
	{



		Situation previous = new Situation();
		Situation current = new Situation();

		Actions previousActionTaken = Actions.NONE;
		Actions actionTaken = Actions.NONE;
		// process 1 agent at a time - so we'll be iterating through
		// the file multiple times
		for (int j=0; j<=numberNodes; j++)
		{
			// reset file to beginning
			try
			{
				in.close();
				in= new BufferedReader(new FileReader(logName));
				// looking for agent j
				// create two situations - previous and current



				// need initial previous context
				// first line should be sim time and buckets moved
				String timeLine = in.readLine();
				// get sim time from parsing the line
				String[] splits = timeLine.split("\t");

				int simTime = Integer.parseInt(splits[1]);



				// second line is number buckets at source
				String numBucketString = in.readLine();
				splits = numBucketString.split("\t");
				int bucketsAtSource = Integer.parseInt(splits[1]);

				// third line is number buckets at sink
				numBucketString = in.readLine();
				splits = numBucketString.split("\t");
				int bucketsAtSink = Integer.parseInt(splits[1]);
				previous = new Situation();
				// now agents
				for (int i=0; i<=numberNodes; i++)
				{
					String agentLine = in.readLine();

					// read line if i==j parse
					if (i==j)
					{

						AgentInfo info = getSituation(agentLine, i);
						info.situation.nodeNum =i;
						previousActionTaken = info.action;
						// place in current situation
						previous.agentStates[i] = info.situation.myState;
						previous.myState = info.situation.myState;
						previous.messageRecvd = 
info.situation.messageRecvd;
						previous.messageSent=info.situation.messageSent;

						previous.bucketsAtSink = bucketsAtSink;
						previous.bucketsAtSource = bucketsAtSource;
						previous.nodeNum = i;
						if (i==0)
						{
							previous.isSourceNode=true;
						} else if (i==numberNodes)
						{
							previous.isRunnerNode = true;
						} else if (i== (numberNodes-1))
						{
							previous.isSinkNode = true;
						}

					}
					else
					{

						AgentInfo tempInfo = getSituation(agentLine, i);
```

```
                                        // just want agent state if one node ahead or behind
current node
                                        if (i==(j-1)  || i==(j+1))
                                        {
                                                previous.agentStates[i] =
tempInfo.situation.agentStates[i];
                                        }
                                }
                        }
                        // done with first entry - train with any possible action
                        Actions[] prevActions = new Actions[1];
                        prevActions[0] = previousActionTaken;
                        trainHashMap(previous, prevActions);

                        // loop until EOF
                        do
                        {
                        // first line should be sim time and buckets moved
                        timeLine = in.readLine();
                        // get sim time from parsing the line
                        splits = timeLine.split("\t");

                        simTime = Integer.parseInt(splits[1]);


                        // second line is number buckets at source
                        numBucketString = in.readLine();
                        splits = numBucketString.split("\t");
                        int sourceBuckets = Integer.parseInt(splits[1]);

                        // third line is number buckets at sink
                        numBucketString = in.readLine();
                        splits = numBucketString.split("\t");
                        int sinkBuckets = Integer.parseInt(splits[1]);
                        current = new Situation();
                        // now agents
                        for (int i=0; i<=numberNodes; i++)
                        {
                                String agentLine = in.readLine();

                                // read line if i==j parse
                                if (i==j)
                                {

                                        AgentInfo info = getSituation(agentLine, i);
                                        info.situation.nodeNum =i;
                                        actionTaken = info.action;
                                        // place in current situation
                                        current.agentStates[i] = info.situation.myState;
                                        current.myState=info.situation.myState;
                                        current.nodeNum =i;
                                        current.messageRecvd =
info.situation.messageRecvd;
                                        current.messageSent = info.situation.messageSent;

                                        current.bucketsAtSink = sinkBuckets;
                                        current.bucketsAtSource = sourceBuckets;
                                        if (i==0)
                                        {
                                             current.isSourceNode=true;
                                        } else if (i==numberNodes)
                                        {
                                             current.isRunnerNode = true;
                                        } else if (i== (numberNodes-1))
                                        {
                                             current.isSinkNode = true;
                                        }
                                   }
                                   else
                                   {
                                        AgentInfo tempInfo = getSituation(agentLine, i);
                                        // just want agent state if one node ahead or behind
current node
                                        if (i==(j-1)  || i==(j+1))
                                        {
                                             current.agentStates[i] =
tempInfo.situation.agentStates[i];
                                        }

                                   }
                              }

                              // gotten all info from file- process current vs
                              // previous
                              if (previous.agentStates[j] == current.agentStates[j])
                              {
                                   // previous context is the same as this one
                                   current.timeInContext = previous.timeInContext+1;
                                   //need new situation to blend previous and current

                                   Situation trainingSituation = new Situation(current);
                                   // make sure node+1 is last state and not transition
```

```
                        if (!trainingSituation.isRunnerNode)
                        {
                                trainingSituation.agentStates[j+1] =
previous.agentStates[j+1];
                                trainingSituation.messageRecvd =
previous.messageRecvd;
                        }
                        trainingSituation.bucketsAtSink = previous.bucketsAtSink;
                        Actions[] trainedActions = new Actions[1];
                        trainedActions[0] = actionTaken;
                        trainHashMap(trainingSituation, trainedActions);

                }
                else
                {
                        // new context - add transition to this context to
                        // previous actions and train
                        current.timeInContext = 0;
                        // previous actions include only previousActionTaken

                        Actions[] trainedActions = new Actions[2];
                        trainedActions[0] = actionTaken;


                        trainedActions[1] = findTransition(current.agentStates[j]);
                   Situation trainingSituation = new Situation(current);
                   trainingSituation.agentStates[j] = previous.agentStates[j];
                   trainingSituation.timeInContext = previous.timeInContext+1;
                   trainingSituation.myState= previous.myState;
                   if (!trainingSituation.isRunnerNode)
                        {
                                trainingSituation.agentStates[j+1] =
previous.agentStates[j+1];
                                trainingSituation.messageRecvd =
previous.messageRecvd;
                        }
                   trainingSituation.bucketsAtSink = previous.bucketsAtSink;
                        trainHashMap(trainingSituation, trainedActions);


                }

                previous = current;
                previousActionTaken = actionTaken;
```

```
				} while (true);

				}   // end try
				catch (EOFException endOfFileException)
				{
					// done with this agent =
					// go on to the next one
					System.out.println("Finished with an agent-EOF");
				}
				catch (Exception ex)
				{
					//ex.printStackTrace();
					System.out.println("Finished with an agent");
				}
			}  // end for (j)

			// done with agents
			// save policies to files
			savePoliciesToFile();


		}


		private void trainHashMap(Situation situation, Actions[] actions)
		{
			HashMap<Situation, Actions[]> myMap = null;
			Vector myVector = null;
			boolean mismatch = false;

			// get map based on context
			int index = situation.myState.ordinal();


			myMap = (HashMap<Situation,Actions[]>)maps.get(index);
			TrainingInfo data = new TrainingInfo();
			data.action = actions;
			data.situation = situation;

			myVector = trainingData[index];
			Set set= myMap.keySet () ;

			//obtain an Iterator for Collection

			Iterator itr = set.iterator();
```

```
//iterate through HashMap values iterator
boolean found = false;
while(itr.hasNext())
{

        Situation latestKey = (Situation)itr.next();
        // check to see if this situation exists in map already
        if (latestKey.equals(situation))
        {
                found = true;
                System.out.println("Got a matching situation");
                // check to see if actions map
                Actions[] savedActions = (Actions[])myMap.get(situation);
                // first check for size map
                if (savedActions.length == actions.length)
                {
                        for (int i =0; i< actions.length; i++)
                        {
                                if (savedActions[i] != actions[i])
                                {
                                        mismatch = true;
                                        break;
                                }
                        }
                }
                else
                {
                        // mismatch in actions
                        mismatch = true;
                }
                // we have mismatch or duplicate mark appropriately
                if (mismatch)
                {
                        // get entry from myVector and increment count
                        System.out.println("GOT A MISMATCH");

                }

        }
}
if (!found)
{
        // this is first time for this situation so just add to map
        myMap.put(situation, actions);
}
```

```
}

private Actions findTransition(States newState)
{
	Actions action = Actions.NONE;
	switch (newState)
	{

	case GRABBING:
		return Actions.TRANS_GRABBING;
	case DIPPING:
		return Actions.TRANS_DIPPING;
	case TURNING_TO_HAND:
		return Actions.TRANS_TURN_HAND;
	case HANDING:
		return Actions.TRANS_HANDING;
	case TURNING_TO_GRAB:
		return Actions.TRANS_TURN_GRAB;
	case DUMPING:
		return Actions.TRANS_DUMPING;
	case WAITING:
		return Actions.TRANS_WAITING;
	case RETURNING:
		return Actions.TRANS_RETURNING;
	case DUMPING_BUCKETS:
		return Actions.TRANS_BUCKET_DUMP;
	case RETURNING_TO_SINK:
		return Actions.TRANS_RETURN_SINK;

	}

	return action;
}

private AgentInfo getSituation(String agentLine, int nodeNum)
{
	AgentInfo newSituation = new AgentInfo();
	newSituation.situation = new Situation();
	String[] splits = agentLine.split("\t");
	// get state
	States state = convertStringToState(splits[1]);

	// get action taken
	newSituation.situation.agentStates[nodeNum]= state;

	newSituation.situation.myState = state;
```

```
                newSituation.action = convertStringToActions(splits[2]);

                // get message Sent
                newSituation.situation.messageSent = convertStringToBoolean(splits[3]);
                if (newSituation.situation.messageSent)
                {
                        newSituation.action = Actions.TAKE_BUCKET;
                }

                // get message received
                newSituation.situation.messageRecvd = convertStringToBoolean(splits[4]);




                return newSituation;
        }



        // Method to convert logged state string to actual state
private States convertStringToState(String state)
{
        States newState = null;

        if (state.equals("GRABBING"))
        {
                return States.GRABBING;
        }
        else if (state.equals("DIPPING"))
                {
                return States.DIPPING;
                }
        else if (state.equals("TURNING_TO_HAND"))
        {
                return States.TURNING_TO_HAND;
        }
        else if (state.equals("HANDING"))
        {
                return States.HANDING;
        }
        else if (state.equals("TURNING_TO_GRAB"))
        {
                return States.TURNING_TO_GRAB;
        }
        else if (state.equals("DUMPING"))
        {
                return States.DUMPING;
        }
        else if (state.equals("WAITING"))
        {
                return States.WAITING;
        }
        else if (state.equals("RETURNING"))
        {
                return States.RETURNING;
        }
        else if (state.equals("DUMPING_BUCKETS"))
        {
                return States.DUMPING_BUCKETS;
        }
        else if (state.equals("RETURNING_TO_SINK"))
        {
                return States.RETURNING_TO_SINK;
        }
        else
        {
                return States.UNKNOWN;
        }
}

// Method to convert logged action string to action
private Actions convertStringToActions(String actionString)
{

        if (actionString.equals("DUMP"))
        {
                return Actions.DUMP;
        } else if (actionString.equals("DIP"))
        {
                return Actions.DIP;
        } else if (actionString.equals("TAKE_BUCKETS"))
        {
                return Actions.TAKE_BUCKETS;
        } else if (actionString.equals("RETURN_BUCKETS"))
        {
                return Actions.RETURN_BUCKETS;
        } else if (actionString.equals("TRANS_GRABBING"))
        {
                return Actions.TRANS_GRABBING;
        } else if (actionString.equals("TRANS_DIPPING"))
        {
                return Actions.TRANS_DIPPING;
        } else if (actionString.equals("TRANS_TURN_HAND"))
        {
                return Actions.TRANS_TURN_HAND;
        } else if (actionString.equals("TRANS_HANDING"))
        {
                return Actions.TRANS_HANDING;
        } else if (actionString.equals("TRANS_TURN_GRAB"))
        {
                return Actions.TRANS_TURN_GRAB;
        } else if (actionString.equals("TRANS_DUMPING"))
        {
                return Actions.TRANS_DUMPING;
        } else if (actionString.equals("TRANS_WAITING"))
        {
                return Actions.TRANS_WAITING;
        } else if (actionString.equals("TRANS_RETURNING"))
        {
                return Actions.TRANS_RETURNING;
        } else if (actionString.equals("TRANS_BUCKET_DUMP"))
        {
                return Actions.TRANS_BUCKET_DUMP;
        } else if (actionString.equals("TRANS_RETURN_SINK"))
        {
                return Actions.TRANS_RETURN_SINK;
        } else
        {
                return Actions.NONE;
        }
}


// method to convert string to boolean
boolean convertStringToBoolean(String boolString)
{
        if (boolString.equals("false"))
        {
                return false;
        }
        else
        {
                return true;
        }
}
```

```
private class AgentInfo
{
        public Situation situation;
        public Actions action;
}

private class TrainingInfo
{
        public Situation situation;
        public Actions[] action = new Actions[2];
        int count = 0;
}


private void savePoliciesToFile()
{
        for (int k=0; k< States.UNKNOWN.ordinal(); k++)
        {
                ObjectOutputStream output = null;
                String fileName = "default.map";
                // get policy hashmap to save
                HashMap<Situation, Actions[]> myMap = null;

                // get map based on context
                myMap = (HashMap<Situation,Actions[]>)maps.get(k);
                // get name of file to save to
                String temp = stateStrings[k];
                fileName = temp+".map";

                // open file for writing
                try // open file
              {
                        output = new ObjectOutputStream(
                           new FileOutputStream( fileName ) );

                        // write to file
                        output.writeObject( myMap );

                        // close file
                        output.close();
              } // end try

                catch (IOException ioException)
              {
                  ioException.printStackTrace();
                System.err.println( "Error opening file." );
              } // end catch

                // open file for writing
                try // open file
              {
                         String txtFileName = temp + ".txt";
                      PrintWriter out = new PrintWriter(
                        new FileWriter( txtFileName ) );

                      Set keySet = myMap.keySet();
                      Iterator iter = keySet.iterator();
                      while (iter.hasNext())
                      {
                         Situation sit = (Situation)iter.next();
                         sit.print(out);
                         Actions[] actions = myMap.get(sit);
                         for (int l=0; l<actions.length; l++)
                         {
                                out.println("Action["+ l+ "] = " + actions[l].toString());
                         }
                      }


                      // close file
                      out.close();
              } // end try

                catch (IOException ioException)
              {
                  ioException.printStackTrace();
                System.err.println( "Error opening file." );
              } // end catch
          } // end for k
      }

          /**
           * @param args
           */
          public static void main(String[] args)
          {
                // get log file name with training data
                // from arguments or constant
                String logFileName = "bucketBrigade.log";
                // create trainer class
                Trainer myTrainer = new Trainer(logFileName);
```

```
        myTrainer.createHashMaps();

        // print names of created hashmap files


    }

}
```

Situation.java

```
package Common;

import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.io.Serializable;


// Class describing the current world state of an agent

//Class describing the current world state of an agent

public class Situation implements Constants, Serializable
{
	public int timeInContext;
	public boolean messageSent;
	public boolean messageRecvd;
	public int bucketsAtSource;
	public int bucketsAtSink;
	public States[] agentStates;
	public boolean isSinkNode;
	public boolean isSourceNode;
	public boolean isRunnerNode;
	public States myState;
	public int nodeNum;



	// default constructor
	public Situation()
	{
		agentStates = new States[Constants.numberNodes +1];
		for (int i=0; i<= Constants.numberNodes; i++)
		{
			agentStates[i]= States.UNKNOWN;
			myState = States.UNKNOWN;
			timeInContext = 0;
			messageSent = false;
			messageRecvd = false;
		}
	}

	// copy constructor
	public Situation(Situation copy)
```

```
{
        agentStates= new States[Constants.numberNodes+1];
        for (int i=0; i<=Constants.numberNodes; i++)
        {
                agentStates[i] = copy.agentStates[i];
        }
        myState = copy.myState;
        timeInContext = copy.timeInContext;
        messageSent = copy.messageSent;
        messageRecvd = copy.messageRecvd;
        bucketsAtSource = copy.bucketsAtSource;
        bucketsAtSink = copy.bucketsAtSink;
        isSinkNode = copy.isSinkNode;
        isSourceNode = copy.isSourceNode;
        isRunnerNode = copy.isRunnerNode;
        nodeNum = copy.nodeNum;
}


public boolean equals(Situation newSituation)
{
        // determine if newSituation is equal to this one
        boolean result = true;
        if (timeInContext != newSituation.timeInContext ||
                messageSent != newSituation.messageSent    ||
                messageRecvd != newSituation.messageRecvd  ||
                bucketsAtSource != newSituation.bucketsAtSource ||
                bucketsAtSink != newSituation.bucketsAtSink ||
                isSinkNode != newSituation.isSinkNode        ||
                isSourceNode != newSituation.isSourceNode  ||
                isRunnerNode != newSituation.isRunnerNode ||
                myState != newSituation.myState  ||
                nodeNum != newSituation.nodeNum)
        {
                result = false;
        }
        else  if (agentStates.length ==newSituation.agentStates.length)
                // check agent States
        {
                for (int i=0; i<agentStates.length; i++)
                {
                        if (agentStates[i]!= newSituation.agentStates[i])
                        {
                                result = false;
                                break;
                        }
```

```
				}
			}
			else // agent states length don't match
			{
				result = false;
			}


			return result;
		}

		public int nearestNeighborScore(Situation compareTo)
		{
			int score = 0;
			if (this.equals(compareTo))
			{
				return score;
			}
			// start with myState - if myState does not match
			// give a very high score
			if (this.myState != compareTo.myState)
			{
				score+=300;
			}

			// type of agent also very important - high score
			// for mismatch
			if (this.isRunnerNode != compareTo.isRunnerNode)
			{
				score+=100;
			}

			if (this.isSinkNode != compareTo.isSinkNode)
			{
				score+=100;
			}

			if (this.isSourceNode != compareTo.isSourceNode)
			{
				score+=100;
			}

			//
			// now compare neighboring node states
			if (!compareTo.isSourceNode && !this.isSourceNode
                && !this.isRunnerNode)
{
        if (this.agentStates[this.nodeNum-1] !=
                compareTo.agentStates[compareTo.nodeNum-1])
        {

                score+=100;
        }
}

if (!compareTo.isSinkNode && !this.isSinkNode
                && !this.isRunnerNode)
{
        if (this.agentStates[this.nodeNum+1] !=
                compareTo.agentStates[compareTo.nodeNum+1]
           && !this.isRunnerNode)
        {

                //score+=50;
        }
}

if (this.isRunnerNode)
{
        if (this.bucketsAtSink < compareTo.bucketsAtSink)
        {
                score+=200;
        }
        else
        {
                score+=Math.abs(compareTo.bucketsAtSink-this.bucketsAtSink);
        }
}

if (this.isSourceNode)
{
        // number of buckets at source is important too
        score+=Math.abs(compareTo.bucketsAtSource-this.bucketsAtSource);
}

// now remaining situation
// messageRecvd
if (this.messageRecvd != compareTo.messageRecvd)
{
        score+=300;
}
```

```
		// timeinContext
		if (this.timeInContext < compareTo.timeInContext)
		{
			score+=100;
		}
		else
		{
			score+=(Math.abs(this.timeInContext-compareTo.timeInContext)*10);
		}

		return score;
	}

	public int contextuallyWeightedNearestNeighbor(Situation compareTo, States context)
	{
		int score = 0;

		if (this.equals(compareTo))
		{
			return score;
		}
		WeightClass wts = getWeights(context);
		int[] weights = wts.weights;

		// start with myState - if myState does not match
		// give a very high score - all contexts
		if (this.myState != compareTo.myState)
		{
			score+=300;
		}

		// type of agent also very important - high score
		// for mismatch
		if (this.isRunnerNode != compareTo.isRunnerNode)
		{
			score+=(100 * weights[0]);
		}

		if (this.isSinkNode != compareTo.isSinkNode)
		{
			score+=(100 * weights[1]);
		}
		if (this.isSourceNode != compareTo.isSourceNode)
		{
			score+=(100 * weights[2]);
		}

		//
		// now compare neighboring node states
		if (!compareTo.isSourceNode && !this.isSourceNode)

		{
			if (this.agentStates[this.nodeNum-1] !=
				compareTo.agentStates[compareTo.nodeNum-1])
			{

				score+=(100 * weights[3]);
			}
		}

		if ( !this.isRunnerNode && !compareTo.isRunnerNode)
		{
			if (this.agentStates[this.nodeNum+1] !=
				compareTo.agentStates[compareTo.nodeNum+1]
			   && !this.isRunnerNode)
			{

				score+=(50 * weights[4]);
			}
		}


		score+=(10 * Math.abs(compareTo.bucketsAtSink-this.bucketsAtSink)*
weights[5]) ;




		// number of buckets at source is important too
		score+=(Math.abs(compareTo.bucketsAtSource-this.bucketsAtSource) *
weights[6]);


		// now remaining situation
		// messageRecvd
		if (this.messageRecvd != compareTo.messageRecvd)
		{
			score+=(300 * weights[7]);
		}



		// timeinContext
		if (this.timeInContext < compareTo.timeInContext)
		{
			score+=(100 * weights[8]);
		}
		else
		{
			score+=((Math.abs(this.timeInContext-compareTo.timeInContext)*10) *
weights[9]);
		}

		return score;
	}

	public void print(PrintWriter out) throws IOException
	{
		// write structure to file
		out.println("********************");

		out.println("MyState: " + myState.toString());
		out.println("Time in Context:" + timeInContext );
		out.println("MessageRecvd: " + messageRecvd);
		out.println("Message Sent: " + messageSent);
		out.println("Buckets At Source: " + bucketsAtSource);
		out.println("Buckets At Sink: "  + bucketsAtSink);
		out.println("Is source: " + isSourceNode);
		out.println("Is Sink : " + isSinkNode) ;
		out.println("Is Runner: " + isRunnerNode);
		out.println("nodeNum : " + nodeNum);
		for (int i=0; i<= Constants.numberNodes; i++)
		{
			out.println("AgentState["+ i+"]= " + agentStates[i].toString());
		}



	}

	private WeightClass getWeights(States context)
	{
		int[] weights = new int[10];
		for (int i=0; i< weights.length; i++)
		{
			weights[i]=0;
		}

		// weights[0] - isRunnerNode
		// for all contexts - weight=1
		weights[0] = 1;

		//weights[1] - isSinkNOde
		weights[1] =1;
		// weights[2] - isSourceNode
		weights[2] = 1;
		// weights[3] - neighbor -1
		if (context == States.GRABBING)
		{
			weights[3] = 1;
		}
		// weights[4] - neighbor +1
		if (context == States.HANDING)
		{
			weights[4]=1;
		}

		// weights[6] - bucketsAtSource
		if (context == States.DIPPING)
		{
			weights[6] =0;
		}

		// weights[5] - bucketsAtSink
		if (context == States.WAITING)
		{
			weights[5] =1;
		}

		// weights[7] - messageRecvd
		if (context == States.HANDING)
		{
			weights[7] = 1;
		}
		// weights[8] - timeInContext < compareTo
		// weights[9] - differencei in time of context
		if (context == States.RETURNING || context == States.TURNING_TO_HAND ||
			context == States.TURNING_TO_GRAB || context ==
States.RETURNING_TO_SINK ||
			context == States.WAITING)
```

```
				{
					weights[8] = 1;
					weights[9] =1;
				}
				WeightClass wts = new WeightClass();
				wts.weights= weights;
				return wts;
			}

			private class WeightClass
			{
				int[] weights;
			}
		}
```

Constants.java

```
package Common;
// Bucket Brigade simulation- Cynthia Johnson

public interface Constants
{
    public enum States  {GRABBING, DIPPING, TURNING_TO_HAND, HANDING,
TURNING_TO_GRAB, DUMPING, WAITING, RETURNING,
        DUMPING_BUCKETS, RETURNING_TO_SINK, UNKNOWN};
    public enum Actions {NONE, DUMP, DIP, TAKE_BUCKET, TAKE_BUCKETS,
RETURN_BUCKETS, TRANS_GRABBING, TRANS_DIPPING, TRANS_TURN_HAND,
        TRANS_HANDING, TRANS_TURN_GRAB, TRANS_DUMPING, TRANS_WAITING,
TRANS_RETURNING, TRANS_BUCKET_DUMP, TRANS_RETURN_SINK};
    public enum MessageTypes { TAKE_BUCKET};
    public final int HAND_TURN_TIME = 1;
    public  final int GRAB_TURN_TIME = 1;
    public final int DIP_TIME =1;
    public final int RUNNING_TIME = 3;
    public final int BUCKETS_TO_RUN = 5;
    public final int numberNodes =6;  // includes source and sink
    public String[] stateStrings=  {"grabbing","dipping",
"turningToHand","handing","turningToGrab",

      "dumping","waiting","returning","dumpingBuckets","returningToSink",
"unknown"};

}
```

APPENDIX B – BUCKET BRIGADE RUN-TIME AGENT CODE

Agent.java

```
import Common.Situation;


public class Agent implements Common.Constants
{

            protected boolean messageRecvd = false;
            protected boolean messageSent = false;
            protected Context context;
            protected int contextStartTime=0;
            protected TeamContext teamContext;
            protected Situation currentSituation;
            protected int nodeNum;
            protected boolean isSinkNode = false;
            protected boolean isSourceNode = false;
            protected boolean isRunnerNode = false;


            public Agent(int nodeNumber, States startingContext)
            {
                        //get TeamContext
                        nodeNum = nodeNumber;
                        teamContext = TeamContext.getInstance();
                        context = teamContext.translateNameToContext(startingContext);
                        if (nodeNum == 0)
                        {
                                    isSourceNode = true;
                        }
                        else if (nodeNum == numberNodes)
                        {
                                    isRunnerNode = true;
                        }
                        else if (nodeNum == (numberNodes-1))
                        {
                                    isSinkNode = true;
                        }
            }

   public void step(int simTime)
   {
            messageSent = false;
            // get situation setup
            currentSituation = teamContext.getAgentInfo(nodeNum);
            // add information from local agent
            currentSituation.messageRecvd = messageRecvd;
            currentSituation.messageSent = messageSent;
            currentSituation.isRunnerNode = isRunnerNode;
            currentSituation.isSinkNode = isSinkNode;
            currentSituation.isSourceNode = isSourceNode;
            currentSituation.timeInContext = simTime-contextStartTime;

            //currentSituation.timeInContext = simTime-contextStartTime;

            // now call current context to determine action
            context.step(this, simTime, currentSituation);
            // clear any messages processed this go around
            messageRecvd = false;
            SimEngine.getInstance().messageRecvd[nodeNum] =false;
    }

    public void receiveMessage(MessageTypes type, int simTime)
    {
            //System.out.println("Received message- node number" + nodeNum);
            messageRecvd = true;
    }

    // methods for context to set situation data
    void setContextStartTime(int time)
    {
            contextStartTime = time;
    }

    void setMessageSent()
    {
            messageSent = true;
    }

    void setNewContext(States newContext)
    {
            context = teamContext.translateNameToContext(newContext);
            teamContext.setNewContext(newContext,nodeNum);
            setContextStartTime(SimEngine.getInstance().getSimTime());
    }

    void sendMessage()
    {

            if (nodeNum-1 >=0)
            {
                    System.out.println("Sent message: Node Num: "+ nodeNum);
                    SimEngine.getInstance().sendMessage(nodeNum-1,       MessageTypes.TAKE_BUCKET,
nodeNum);
                    messageSent = true;

            }
    }
```

```
}
```

Context.java

```
import java.io.EOFException;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.util.Collection;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Set;


import Common.Constants;
import Common.Situation;


public  class Context implements Common.Constants
{
            protected String name;
            protected HashMap<Situation, Constants.Actions[]>   policy;
            protected int nodeNum;

            public Context(String contextName)
            {
                        name= contextName;
                        initializePolicy(name+".map");
            }


            public void step(Agent owningAgent,int simTime, Situation currentSituation)
            {
                        nodeNum=currentSituation.nodeNum;
                        // check Hashmap for current situation

                                    // find nearest neighbor

                                    /*
                                    #get Collection of keys contained in HashMap using
                                    #  Collection values() method of HashMap class
                                    # */

                                     Set set= policy.keySet () ;

                                    //obtain an Iterator for Collection

                                    Iterator itr = set.iterator();

                                    //iterate through HashMap values iterator
                                    Actions[] actions = null;
           int lowScore = 1000;
           Situation lowScorer = null;
           int count = 0;
```

```
        if (currentSituation.myState == States.WAITING && currentSituation.bucketsAtSink >=3)
        {
         System.out.println("Waiting");
        }

                        while(itr.hasNext())
                        {
                            Situation mySituation = (Situation)itr.next();
                            //int score = currentSituation.nearestNeighborScore(mySituation);
                            if (mySituation.bucketsAtSink > 3)
                            {
                                int i = 2;
                            }

                            int                              score                              =
    currentSituation.contextuallyWeightedNearestNeighbor(mySituation, currentSituation.myState);

                            if (score < lowScore)
                            {
                                lowScore = score;
                                lowScorer= mySituation;
                                actions = (Actions[])policy.get(lowScorer);
                            }
                            count ++;
                        }

                        // set anything necessary in calling Agent
                        for (int i=0; i<actions.length;i++)
                        {
                            Actions action = actions[i];

                            performAction(action, simTime, owningAgent);
                        }

                        if (lowScorer.messageSent)
                        {
                            //System.out.println("low scorer sent message");
                            owningAgent.sendMessage();

                        }
                    }



         private void performAction(Actions action, int simTime, Agent owningAgent)
         {
                    switch (action)
                    {
                    case DUMP:
                        // dump water into sink
                        SimEngine.getInstance().dumpBucket(owningAgent.nodeNum);
                        break;

                    case DIP:
                        // take water and bucket from source
                        SimEngine.getInstance().takeBucketFromSource(owningAgent.nodeNum);
                        break;

                    case TAKE_BUCKET:
                        // send message to node-1 that we are reliving them of
                        // bucket
                        SimEngine.getInstance().sendMessage(nodeNum-1,
    MessageTypes.TAKE_BUCKET, nodeNum);
                        break;

                    case TAKE_BUCKETS:
                        // runner takes five buckets from sink to return
                        // to source
                        SimEngine.getInstance().takeBucketsAtSink(5, owningAgent.nodeNum);
                        break;

                    case RETURN_BUCKETS:
                        // runner returns the five buckets to the source
                        SimEngine.getInstance().addBucketsAtSource(5, owningAgent.nodeNum);
                        break;

                    case TRANS_GRABBING:
                        // transition context to grabbing
                        owningAgent.setNewContext(States.GRABBING);
                        owningAgent.setContextStartTime(simTime);
                        break;

                    case TRANS_DIPPING:
                        // transition context to grabbing
                        owningAgent.setNewContext(States.DIPPING);
                        owningAgent.setContextStartTime(simTime);
                        break;

                    case TRANS_TURN_HAND:
                        // transition context to grabbing
                        owningAgent.setNewContext(States.TURNING_TO_HAND);
                        owningAgent.setContextStartTime(simTime);
                        break;

                    case TRANS_HANDING:
                        // transition context to grabbing
                        owningAgent.setNewContext(States.HANDING);
                        owningAgent.setContextStartTime(simTime);
                        break;

                    case TRANS_TURN_GRAB:
                        // transition context to grabbing
```

```
                        owningAgent.setNewContext(States.TURNING_TO_GRAB);
                        owningAgent.setContextStartTime(simTime);
                        break;

                case TRANS_DUMPING:
                        // transition context to grabbing
                        owningAgent.setNewContext(States.DUMPING);
                        owningAgent.setContextStartTime(simTime);
                        break;

                case TRANS_WAITING:
                        // transition context to grabbing
                        owningAgent.setNewContext(States.WAITING);
                        owningAgent.setContextStartTime(simTime);
                        break;

                case TRANS_RETURNING:
                        // transition context to grabbing
                        owningAgent.setNewContext(States.RETURNING);
                        owningAgent.setContextStartTime(simTime);
                        break;

                case TRANS_BUCKET_DUMP:
                        // transition context to grabbing
                        owningAgent.setNewContext(States.DUMPING_BUCKETS);
                        owningAgent.setContextStartTime(simTime);
                        break;

                case TRANS_RETURN_SINK:
                        // transition context to grabbing
                        owningAgent.setNewContext(States.RETURNING_TO_SINK);
                        owningAgent.setContextStartTime(simTime);
                        break;
   default:
                 // do nothing
                 // break;
                }
}



private void initializePolicy(String fileName)
{
                ObjectInputStream input = null;
                try // open file
      {
         input = new ObjectInputStream(
            new FileInputStream( fileName ) );
      } // end try
      catch ( IOException ioException )
      {
```

```
                System.err.println( "Error opening file." );
              } // end catch

                    // read in single HashMap of policy generated by
              // learning algorithm
              try // input the values from the file
              {
                     Object tempPolicy = input.readObject();
                     if (tempPolicy instanceof HashMap<?,?>)
                     {
                           policy = (HashMap<Situation, Constants.Actions[]>)tempPolicy;
                     }


              } // end try
              catch ( EOFException endOfFileException )
              {
               System.out.println("end of file reached- nothing read");
              } // end catch
              catch ( ClassNotFoundException classNotFoundException )
              {
                    classNotFoundException.printStackTrace();
                System.err.println( "Unable to create object." );
              } // end catch
              catch ( IOException ioException )
              {
                    ioException.printStackTrace();
                System.err.println( "Error during reading from file." );
              } // end catch

          }
  }
```

TeamContext.java

```
import Common.Constants;
import Common.Situation;

// team context for CCXBR framework
// author: Cynthia Johnson
public class TeamContext implements Common.Constants
{
      private static boolean initialized = false;
      private static TeamContext instance = null;
      private SimEngine simEngine;

      // initialize all the contexts in this game
      private                Context                grabbing=                new
Context(stateStrings[States.GRABBING.ordinal()]);
      private             Context            dipping             =            new
Context(stateStrings[States.DIPPING.ordinal()]);
      private            Context          turningToHand          =            new
Context(stateStrings[States.TURNING_TO_HAND.ordinal()]);
      private             Context            handing             =            new
Context(stateStrings[States.HANDING.ordinal()]);
      private            Context          turningToGrab          =            new
Context(stateStrings[States.TURNING_TO_GRAB.ordinal()]);
      private             Context            dumping             =            new
Context(stateStrings[States.DUMPING.ordinal()]);
      private             Context            waiting             =            new
Context(stateStrings[States.WAITING.ordinal()]);
      private             Context           returning            =            new
Context(stateStrings[States.RETURNING.ordinal()]);
      private            Context          dumpingBuckets          =           new
Context(stateStrings[States.DUMPING_BUCKETS.ordinal()]);
      private            Context          returningToSink         =           new
Context(stateStrings[States.RETURNING_TO_SINK.ordinal()]);


      private TeamContext()
      {
            // get a link to sim engine to build situations
            // for various agents
            simEngine = SimEngine.getInstance();
      }
      public static TeamContext getInstance()
      {
            // method to allow agents to get instance of team context
            if (!initialized)
            {
                  instance = new TeamContext();
                  initialized = true;
            }

            return instance;
      }

      public Situation getAgentInfo(int node)
      {
            // returns the situation for the agent at
            // node numbered node.
            Situation situation = new Situation();
            // fill information in the situation
            // note that agents only actually known context
            // of nodes to either side of them
            simEngine=SimEngine.getInstance();
            situation.bucketsAtSink=simEngine.getBucketsAtSink();
            situation.bucketsAtSource = simEngine.getBucketsAtSource();
            if ((node -1) >= 0)
            {
                  situation.agentStates[node-1]                               =
simEngine.getAgentState(node-1);
            }

            if ((node +1) <= Constants.numberNodes)
            {

      situation.agentStates[node+1]=simEngine.getAgentState(node+1);
            }

          situation.agentStates[node] = simEngine.getAgentState(node);
          situation.myState = situation.agentStates[node];
          situation.nodeNum = node;

            return situation;
      }

      Context translateNameToContext(States contextName)
      {
            switch (contextName)
            {
            case GRABBING:
                  return grabbing;

            case DIPPING:
                  return dipping;

            case TURNING_TO_HAND:
                  return turningToHand;

            case HANDING:
                  return handing;

            case TURNING_TO_GRAB:
                  return turningToGrab;

            case DUMPING:
                  return dumping;

            case WAITING:
                  return waiting;

            case RETURNING:
                  return returning;

            case DUMPING_BUCKETS:
                  return dumpingBuckets;
```

```
			case RETURNING_TO_SINK:
				return returningToSink;

			default:
				return null;
			}
		}

		public void setNewContext(States newContext, int nodeNum)
		{
			SimEngine.getInstance().setNodeState(nodeNum, newContext);
		}


	}
```

APPENDIX C- PURSUIT GAME TRAINING CODE

```
Trainer.java
package Training;
// main class for training algorithm for bucket brigade
import Common.Constants;
import Common.Situation;
import Common.Position;


import java.io.BufferedReader;
import java.io.EOFException;
import java.io.FileReader;
import java.io.FileOutputStream;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;

import java.io.ObjectOutputStream;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Iterator;
import java.util.List;
import java.util.Map;
import java.util.Set;
import java.util.Vector;


public class Trainer implements Constants
{
        private BufferedReader in;
        private String logName = null;
        private List<Map<Situation, Actions[]>> maps = new ArrayList<Map<Situation, Actions[]>>();
        private Vector[] trainingData = new Vector[14];
   private final int TRANS_GO_TO_DISTANCE = 5;


        public Trainer(String fileName)
        {

           logName = fileName;
                try
                {
                // open file for reading
                        in= new BufferedReader(new FileReader(logName));
                }
                catch (Exception ex)
                {
                        System.out.println("Error opening log file");
                        ex.printStackTrace();
                }

                //create a hash map for each context
```

```
		for (int i=0; i<= 13; i++)
		{
			maps.add(new HashMap<Situation, Actions[]>());
			trainingData[i] = new Vector();
		}

	}

	public void createHashMaps()
	{


		Situation previous = new Situation();
		Situation current = new Situation();

		int previousSimTime =0;
		int currentSimTime=0;


		// process 1 agent at a time - so we'll be iterating through
		// the file multiple times
		for (int j=0; j<=4; j++)
		{
			// reset file to beginning
			try
			{

				in.close();
				in= new BufferedReader(new FileReader(logName));
				// looking for agent j
				// create two situations - previous and current

				previous = new Situation();
				// need initial previous context
				// first line should be sim time and buckets moved
				String timeLine = in.readLine();
				// get sim time from parsing the line
				String[] splits = timeLine.split("\t");

				int simTime = Integer.parseInt(splits[1]);

				previousSimTime = simTime;
				// second line is blue1 position
				String blue1String = in.readLine();
				splits = blue1String.split("\t");
				Position blue1Position = new Position();
				blue1Position.row = Integer.parseInt(splits[1]);
				blue1Position.column = Integer.parseInt(splits[2]);
				previous.blue1Position= blue1Position;
				// Knows red position
				blue1String = in.readLine();
```

```
splits = blue1String.split("\t");
Boolean blue1KnowsRed = Boolean.parseBoolean(splits[1]);
previous.redPositionKnown=blue1KnowsRed.booleanValue();


// blue2
String blue2String = in.readLine();
splits = blue2String.split("\t");
Position blue2Position = new Position();
blue2Position.row = Integer.parseInt(splits[1]);
blue2Position.column = Integer.parseInt(splits[2]);
previous.blue2Position= blue2Position;
// Knows red position
blue2String = in.readLine();
splits = blue2String.split("\t");
Boolean blue2KnowsRed = Boolean.parseBoolean(splits[1]);
if (previous.redPositionKnown == false )
{
        previous.redPositionKnown=blue2KnowsRed.booleanValue();
}

// blue3
String blue3String = in.readLine();
splits = blue3String.split("\t");
Position blue3Position = new Position();
blue3Position.row = Integer.parseInt(splits[1]);
blue3Position.column = Integer.parseInt(splits[2]);
previous.blue3Position= blue3Position;
// Knows red position
blue3String = in.readLine();
splits = blue3String.split("\t");
Boolean blue3KnowsRed = Boolean.parseBoolean(splits[1]);
if (previous.redPositionKnown == false )
{
        previous.redPositionKnown=blue3KnowsRed.booleanValue();
}

// blue4
String blue4String = in.readLine();
splits = blue4String.split("\t");
Position blue4Position = new Position();
blue4Position.row = Integer.parseInt(splits[1]);
blue4Position.column = Integer.parseInt(splits[2]);
previous.blue4Position= blue4Position;
// Knows red position
blue4String = in.readLine();
splits = blue3String.split("\t");
Boolean blue4KnowsRed = Boolean.parseBoolean(splits[1]);
if (previous.redPositionKnown == false )
{
        previous.redPositionKnown=blue4KnowsRed.booleanValue();
```

```
}

// red
String redString = in.readLine();
splits = redString.split("\t");
Position redPosition = new Position();
redPosition.row = Integer.parseInt(splits[1]);
redPosition.column = Integer.parseInt(splits[2]);
previous.redPosition= redPosition;

if (j==0)
{
        // we are blue1
        previous.myPosition=previous.blue1Position;
        previous.myType = PlayerType.BLUE1;
        // starting context is search top left - only set here
        previous.myState = States.SEARCH_TOP_LEFT;
}
else if (j==1)
{
        // we are blue2
        previous.myPosition=previous.blue2Position;
        previous.myType = PlayerType.BLUE2;
        // starting context is search top right - only set here
        previous.myState = States.SEARCH_TOP_RIGHT;
}
else if (j==2)
{
        // we are blue3
        previous.myPosition=previous.blue3Position;
        previous.myType = PlayerType.BLUE3;
        // starting context is search top right - only set here
        previous.myState = States.SEARCH_BOT_LEFT;
}
else if (j==3)
{
        // we are blue4
        previous.myPosition=previous.blue4Position;
        previous.myType = PlayerType.BLUE4;
        // starting context is search top right - only set here
        previous.myState = States.SEARCH_BOT_RIGHT;
}

// loop until EOF
do
{

//      create current situation
        current = new Situation();
        // need initial previous context
        // first line should be sim time and buckets moved
```

```
                                        timeLine = in.readLine();
                                        // get sim time from parsing the line
                                        splits = timeLine.split("\t");

                                        simTime = Integer.parseInt(splits[1]);

                                        currentSimTime = simTime;
                                        // second line is blue1 position
                                        blue1String = in.readLine();
                                        splits = blue1String.split("\t");
                                        blue1Position = new Position();
                                        blue1Position.row = Integer.parseInt(splits[1]);
                                        blue1Position.column = Integer.parseInt(splits[2]);
                                        current.blue1Position= blue1Position;
                                        // Knows red position
                                        blue1String = in.readLine();
                                        splits = blue1String.split("\t");
                                        blue1KnowsRed = Boolean.parseBoolean(splits[1]);
                                        current.redPositionKnown=blue1KnowsRed.booleanValue();


                                        // blue2
                                        blue2String = in.readLine();
                                        splits = blue2String.split("\t");
                                        blue2Position = new Position();
                                        blue2Position.row = Integer.parseInt(splits[1]);
                                        blue2Position.column = Integer.parseInt(splits[2]);
                                        current.blue2Position= blue2Position;
                                        // Knows red position
                                        blue2String = in.readLine();
                                        splits = blue2String.split("\t");
                                        blue2KnowsRed = Boolean.parseBoolean(splits[1]);
                                        if (current.redPositionKnown == false )
                                        {

                current.redPositionKnown=blue2KnowsRed.booleanValue();
                                        }

                                        // blue3
                                        blue3String = in.readLine();
                                        splits = blue3String.split("\t");
                                        blue3Position = new Position();
                                        blue3Position.row = Integer.parseInt(splits[1]);
                                        blue3Position.column = Integer.parseInt(splits[2]);
                                        current.blue3Position= blue3Position;
                                        // Knows red position
                                        blue3String = in.readLine();
                                        splits = blue3String.split("\t");
                                        blue3KnowsRed = Boolean.parseBoolean(splits[1]);
                                        if (current.redPositionKnown == false )
                                        {
```

```
	current.redPositionKnown=blue3KnowsRed.booleanValue();
					}

					// blue4
					blue4String = in.readLine();
					splits = blue4String.split("\t");
					blue4Position = new Position();
					blue4Position.row = Integer.parseInt(splits[1]);
					blue4Position.column = Integer.parseInt(splits[2]);
					current.blue4Position= blue4Position;
					// Knows red position
					blue4String = in.readLine();
					splits = blue3String.split("\t");
					blue4KnowsRed = Boolean.parseBoolean(splits[1]);
					if (current.redPositionKnown == false )
					{

	current.redPositionKnown=blue4KnowsRed.booleanValue();
					}

					// red
					redString = in.readLine();
					splits = redString.split("\t");
					redPosition = new Position();
					redPosition.row = Integer.parseInt(splits[1]);
					redPosition.column = Integer.parseInt(splits[2]);
					current.redPosition= redPosition;

					if (j==0)
					{
						// we are blue1
						current.myPosition=current.blue1Position;
						current.myType = PlayerType.BLUE1;
						if (blue1KnowsRed.booleanValue())
						{
							// calculate distance between blue and red
							int distance =  Math.abs(current.myPosition.row
- current.redPosition.row)
							  +     Math.abs(current.myPosition.column     -
current.redPosition.column);
							if (distance > TRANS_GO_TO_DISTANCE)
							{
								current.myState                          =
States.INTERCEPT_TOP;
							}
							else
							{
								current.myState = States.GO_TO_TOP;
							}
```

```
                                        }
                                        else
                                                current.myState = States.SEARCH_TOP_LEFT;
                                }
                                else if (j==1)
                                {
                                        // we are blue2
                                        current.myPosition=current.blue2Position;
                                        current.myType = PlayerType.BLUE2;
                                        if (blue2KnowsRed.booleanValue())
                                        {
                                                int distance =  Math.abs(current.myPosition.row
- current.redPosition.row)
                                                  +     Math.abs(current.myPosition.column      -
current.redPosition.column);
                                                if (distance > TRANS_GO_TO_DISTANCE)
                                                {
                                                        current.myState                          =
States.INTERCEPT_RIGHT;
                                                }
                                                else
                                                {
                                                        current.myState                          =
States.GO_TO_RIGHT;
                                                }
                                        }
                                        else
                                                current.myState                                  =
States.SEARCH_TOP_RIGHT;
                                }
                                else if (j==2)
                                {
                                        // we are blue3
                                        current.myPosition=current.blue3Position;
                                        current.myType = PlayerType.BLUE3;
                                        if (blue3KnowsRed.booleanValue())
                                        {
                                                int distance =  Math.abs(current.myPosition.row
- current.redPosition.row)
                                                  +     Math.abs(current.myPosition.column      -
current.redPosition.column);
                                                if (distance > TRANS_GO_TO_DISTANCE)
                                                {
                                                        current.myState                          =
States.INTERCEPT_LEFT;
                                                }
                                                else
                                                {
                                                        current.myState                          =
States.GO_TO_LEFT;
                                                }
                                        }
                                        else
                                        {
                                                current.myState                              =
States.SEARCH_BOT_LEFT;
                                                if (current.myPosition.row ==0)
                                                {
                                                        System.out.println("Blue       3       upper
found");
                                                }

                                        }
                                }
                                else if (j==3)
                                {
                                        // we are blue4
                                        current.myPosition=current.blue4Position;
                                        current.myType = PlayerType.BLUE4;
                                        if (blue4KnowsRed.booleanValue())
                                        {
                                                int distance =  Math.abs(current.myPosition.row
- current.redPosition.row)
                                                  +     Math.abs(current.myPosition.column      -
current.redPosition.column);
                                                if (distance > TRANS_GO_TO_DISTANCE)
                                                {
                                                        current.myState                              =
States.INTERCEPT_BOTTOM;
                                                }
                                                else
                                                {
                                                        current.myState                              =
States.GO_TO_BOTTOM;
                                                }
                                        }
                                        else
                                                current.myState                              =
States.SEARCH_BOT_RIGHT;
                                }


                        // gotten all info from file- process current vs
                        // previous
                        // calculate what move was made by agent we are currently
                        // training for
                        Actions[] trainedActions = new Actions[3];
                        for (int i=0; i<3;i++)
                        {
                                trainedActions[i] = Actions.NONE;
                        }
                        if (j==0)  // compare blue 1
                        {

                          trainedActions[0]        =        calculateMove(previous.blue1Position,
current.blue1Position);
                        }
                        else if (j==1)  // compare blue 2
                        {
                              trainedActions[0]    =    calculateMove(previous.blue2Position,
current.blue2Position);
                        }
                        else if (j==2)  // compare blue 3
                        {
                              trainedActions[0]    =    calculateMove(previous.blue3Position,
current.blue3Position);
                        }
                        else if (j==3)  // compare blue 4
                        {
                              trainedActions[0]    =    calculateMove(previous.blue4Position,
current.blue4Position);
                        }

                        // add on context switches and message sent


                        if (previous.myState != current.myState )
                        {

                              // new context - add transition to this context to
                              // movement action and check for message sent

                              trainedActions[1]        =        findTransition(previous.myState,
current.myState);



                              // check to see if message sent
                              if (previous.redPositionKnown != current.redPositionKnown &&
                                          current.redPositionKnown)
                              {
                                    trainedActions[2]                                    =
Actions.NOTIFY_RED_POSITION;
                              }


                        }
                        // do not train transition from game to game
                        if (previousSimTime < currentSimTime)
                        {
                              Situation trainingSituation = new Situation(previous);

                              // red moves before any blue so old blue goes with new red
```

```
                                trainingSituation.redPositionKnown = current.redPositionKnown;
                                trainingSituation.redPosition = current.redPosition;
                                if (trainingSituation != null)
                                {
                                        trainHashMap(trainingSituation, trainedActions);
                                }
                                else
                                {
                                        System.out.println("Trained situation = null");
                                }
                            }
                            previous = current;
                            previousSimTime = currentSimTime;



                    } while (true);

                    }   // end try
                    catch (EOFException endOfFileException)
                    {
                            // done with this agent =
                            // go on to the next one
                            System.out.println("Finished with an agent-EOF");
                    }
                    catch (Exception ex)
                    {
                            //ex.printStackTrace();
                            System.out.println("Finished with an agent");
                    }
            }  // end for (j)

            // done with agents
            // save policies to files
            savePoliciesToFile();


    }


    private void trainHashMap(Situation situation, Actions[] actions)
    {
            HashMap<Situation, Actions[]> myMap = null;
            Vector myVector = null;
            boolean mismatch = false;

            // get map based on context
            int index = situation.myState.ordinal();


            myMap = (HashMap<Situation,Actions[]>)maps.get(index);
```

```
TrainingInfo data = new TrainingInfo();
data.action = actions;
data.situation = situation;

myVector = trainingData[index];
Set set= myMap.keySet () ;

//obtain an Iterator for Collection

Iterator itr = set.iterator();

//iterate through HashMap values iterator
boolean found = false;
while(itr.hasNext())
{

        Situation latestKey = (Situation)itr.next();
        // check to see if this situation exists in map already
        if (latestKey.equals(situation))
        {
                found = true;
                //System.out.println("Got a matching situation");
                // check to see if actions map
                Actions[] savedActions = (Actions[])myMap.get(latestKey);
                // first check for size map
                if (savedActions == null)
                {
                        System.out.println("Got null saved actions");
                }
                else if (actions == null)
                {
                        System.out.println("actions = null");

                }
                else if (savedActions.length == actions.length)
                {
                        for (int i =0; i< actions.length; i++)
                        {
                                if (savedActions[i] != actions[i])
                                {
                                        mismatch = true;
                                        System.out.println("Mismatch!");
                                        break;
                                }
                        }
                }
                else
                {
                        // mismatch in actions
                        mismatch = true;
                        System.out.println("Got a mismatched situation");
                        }
                        // we have mismatch or duplicate mark appropriately
                        if (mismatch)
                        {
                                // get entry from myVector and increment count
                                System.out.println("GOT A MISMATCH");

                        }

                }
        }
        if (!found)
        {
                // this is first time for this situation so just add to map
                myMap.put(situation, actions);
        }

}

private Actions findTransition(States previous, States current)
{
        Actions action = Actions.NONE;
        switch (current)
        {

        case SEARCH_TOP_LEFT:
                return Actions.TRANS_SEARCH_TOP_LEFT;
        case SEARCH_TOP_RIGHT:
                return Actions.TRANS_SEARCH_TOP_RIGHT;
        case SEARCH_BOT_LEFT:
                return Actions.TRANS_SEARCH_BOT_LEFT;
        case SEARCH_BOT_RIGHT:
                return Actions.TRANS_SEARCH_BOT_RIGHT;
        case GO_TO_TOP:
                return Actions.TRANS_GO_TO_TOP;
        case GO_TO_RIGHT:
                return Actions.TRANS_GO_TO_RIGHT;
        case GO_TO_LEFT:
                return Actions.TRANS_GO_TO_LEFT;
        case GO_TO_BOTTOM:
                return Actions.TRANS_GO_TO_BOTTOM;
        case INTERCEPT_RIGHT:
                return Actions.TRANS_INTERCEPT_RIGHT;
        case INTERCEPT_LEFT:
                return Actions.TRANS_INTERCEPT_LEFT;
        case INTERCEPT_TOP:
                return Actions.TRANS_INTERCEPT_TOP;
        case INTERCEPT_BOTTOM:
                return Actions.TRANS_INTERCEPT_BOTTOM;

        }
```

```
			return action;
		}




		// Method to convert logged state string to actual state
	private States convertStringToState(String state)
	{
		States newState = null;


		if (state.equals("SEARCH_TOP_LEFT"))
		{
			return States.SEARCH_TOP_LEFT;
		}
		else if (state.equals("SEARCH_TOP_RIGHT"))
			{
			return States.SEARCH_TOP_RIGHT;
			}
		else if (state.equals("SEARCH_BOT_LEFT"))
		{
			return States.SEARCH_BOT_LEFT;
		}
		else if (state.equals("SEARCH_BOT_RIGHT"))
		{
			return States.SEARCH_BOT_RIGHT;
		}
		else if (state.equals("GO_TO_TOP"))
		{
			return States.GO_TO_TOP;
		}
		else if (state.equals("GO_TO_RIGHT"))
		{
			return States.GO_TO_RIGHT;
		}
		else if (state.equals("GO_TO_LEFT"))
		{
			return States.GO_TO_LEFT;
		}
		else if (state.equals("GO_TO_BOTTOM"))
		{
			return States.GO_TO_BOTTOM;
		}
		else
		{
			return States.UNKNOWN;
		}
	}
```

```
// Method to convert logged action string to action
private Actions convertStringToActions(String actionString)
{

        if (actionString.equals("UP"))
        {
                return Actions.UP;
        } else if (actionString.equals("DOWN"))
        {
                return Actions.DOWN;
        } else if (actionString.equals("LEFT"))
        {
                return Actions.LEFT;
        } else if (actionString.equals("RIGHT"))
        {
                return Actions.RIGHT;
        } else
        {
                return Actions.NONE;
        }


}


// method to convert string to boolean
boolean convertStringToBoolean(String boolString)
{
        if (boolString.equals("false"))
        {
                return false;
        }
        else
        {
                return true;
        }
}

private Actions calculateMove(Position previous, Position current)
{
        Actions returnAction = Actions.NONE;

        if (previous.row == current.row)
        {
                // check column
                if (previous.column < current.column)
                {
                        returnAction = Actions.RIGHT;
                }
                else if (previous.column > current.column)
                {
                        returnAction = Actions.LEFT;
                }
        } else if (previous.row < current.row)
        {
                returnAction = Actions.DOWN;
        }
        else if (previous.row > current.row)
        {
                returnAction = Actions.UP;
        }


        return returnAction;
}

private class AgentInfo
{
        public Situation situation;
        public Actions action;
}

private class TrainingInfo
{
        public Situation situation;
        public Actions[] action = new Actions[3];
        int count = 0;
}


private void savePoliciesToFile()
{
        for (int k=0; k< States.UNKNOWN.ordinal(); k++)
        {
                ObjectOutputStream output = null;
                String fileName = "default.map";
                // get policy hashmap to save
                HashMap<Situation, Actions[]> myMap = null;

                // get map based on context
                myMap = (HashMap<Situation,Actions[]>)maps.get(k);
                // get name of file to save to
                String temp = stateStrings[k];
                fileName = temp+".map";

                // open file for writing
                try // open file
            {
                        output = new ObjectOutputStream(
                          new FileOutputStream( fileName ) );

                        // write to file
                output.writeObject( myMap );

                // close file
                output.close();
        } // end try

           catch (IOException ioException)
        {
              ioException.printStackTrace();
          System.err.println( "Error opening file." );
        } // end catch

           // open file for writing
           try // open file
        {
                   String txtFileName = temp + ".txt";
                PrintWriter out = new PrintWriter(
                  new FileWriter( txtFileName ) );

                Set keySet = myMap.keySet();
                Iterator iter = keySet.iterator();
                while (iter.hasNext())
                {
                   Situation sit = (Situation)iter.next();
                   sit.print(out);
                   Actions[] actions = myMap.get(sit);
                   for (int l=0; l<actions.length; l++)
                   {
                        out.println("Action["+ l+ "] = " + actions[l].toString());
                   }
                }


                // close file
                out.close();
        } // end try

           catch (IOException ioException)
        {
              ioException.printStackTrace();
          System.err.println( "Error opening file." );
        } // end catch
    } // end for k
}

    /**
     * @param args
     */
    public static void main(String[] args)
    {
           // get log file name with training data
```

```
                // from arguments or constant
                String logFileName = "pursuitGame.log.train";
                // create trainer class
                Trainer myTrainer = new Trainer(logFileName);

                myTrainer.createHashMaps();




        }

}
```

Situation.java

```
package Common;

import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.io.Serializable;


// Class describing the current world state of an agent

//Class describing the current world state of an agent

public class Situation implements Constants, Serializable
{
	public States myState;
	public PlayerType myType;
	public Position myPosition;
	public Position blue1Position;
	public Position blue2Position;
	public Position blue3Position;
	public Position blue4Position;
	public boolean redPositionKnown;
	public Position redPosition;
	public TeamStates teamState;

	private final int MY_POSITION_WEIGHT =0;
	private final int BLUE1_POSITION_WEIGHT = 2;
	private final int BLUE2_POSITION_WEIGHT =3;
	private final int BLUE3_POSITION_WEIGHT = 4;
	private final int BLUE4_POSITION_WEIGHT =5;
	private final int RED_POSITION_KNOWN_WEIGHT = 6;
	private final int RED_POSITION_WEIGHT =7;
	private final int SLOPE_TO_RED_WEIGHT =8;
	private final int DISTANCE_TO_RED_WEIGHT =9;
	private final int SLOPE_TO_BLUE1_WEIGHT =10;
	private final int SLOPE_TO_BLUE2_WEIGHT =11;
	private final int SLOPE_TO_BLUE3_WEIGHT =12;
	private final int SLOPE_TO_BLUE4_WEIGHT =13;
	private final int DISTANCE_TO_BLUE1_WEIGHT = 14;
	private final int DISTANCE_TO_BLUE2_WEIGHT = 15;
	private final int DISTANCE_TO_BLUE3_WEIGHT = 16;
	private final int DISTANCE_TO_BLUE4_WEIGHT = 17;


	// default constructor
	public Situation()
	{
		redPositionKnown = false;
	}
```

```
// copy constructor
public Situation(Situation copy)
{
        myType = copy.myType;
        myPosition = copy.myPosition;
        blue1Position = copy.blue1Position;
        blue2Position = copy.blue2Position;
        blue3Position = copy.blue3Position;
        blue4Position = copy.blue4Position;
        redPositionKnown = copy.redPositionKnown;
        redPosition= copy.redPosition;
        myState = copy.myState;
        teamState = copy.teamState;
}


public boolean equals(Situation newSituation)
{
        // determine if newSituation is equal to this one
        boolean result = false;
        if (myPosition.equals(newSituation.myPosition)  &&
           blue1Position.equals(newSituation.blue1Position)   &&
           blue2Position.equals(newSituation.blue2Position)  &&
           blue3Position.equals(newSituation.blue3Position)  &&
           blue4Position.equals(newSituation.blue4Position)  &&
           redPositionKnown == newSituation.redPositionKnown &&
           myState == newSituation.myState &&
           teamState == newSituation.teamState)
        {
                if (redPosition != null && redPosition.equals(newSituation.redPosition)
                                || (redPosition==null &&  newSituation.redPosition== null))
                {
                        result = true;
                }
        }


        return result;
}



public int contextuallyWeightedNearestNeighbor(Situation compareTo, States context)
{
        double score = 0;

        if (this.equals(compareTo))
        {
                return (int)score;
        }
        WeightClass wts = getWeights(context);
```

```
double[] weights =wts.weights;

// start with myState - if myState does not match
// give a very high score - all contexts
// shouldn't happen unless context is wrong
if (this.myState != compareTo.myState)
{
        score+=300;
}

//my position
int temp = myPosition(this).difference(myPosition(compareTo));
score+= (temp*weights[MY_POSITION_WEIGHT]);


if (this.redPositionKnown != compareTo.redPositionKnown)
{
        score= score + (int)(weights[RED_POSITION_KNOWN_WEIGHT]);
}

// blue1 position
temp = this.blue1Position.difference(compareTo.blue1Position);
score+= (temp*weights[BLUE1_POSITION_WEIGHT]);
// blue2 slope
double t1 = getSlope(this.blue1Position, this.myPosition);
double t2 = getSlope(compareTo.blue1Position, compareTo.myPosition);
score= score +(Math.abs(t1-t2)*weights[SLOPE_TO_BLUE1_WEIGHT]);
// compare distance between myPosition and blue1Position
double myDistance = Math.abs(myPosition.row-blue1Position.row) +
   Math.abs(myPosition.column-blue1Position.column);
double compareDistance = Math.abs(compareTo.myPosition.row -
                compareTo.blue1Position.row) + Math.abs(
                compareTo.myPosition.column - compareTo.blue1Position.column);
double difference = Math.abs(myDistance-compareDistance);
score = score + (Math.abs(difference)*weights[DISTANCE_TO_BLUE1_WEIGHT]);


// blue2 position
temp = this.blue2Position.difference(compareTo.blue2Position);
score+= (temp*weights[BLUE2_POSITION_WEIGHT]);
t1 = getSlope(this.blue2Position, this.myPosition);
t2 = getSlope(compareTo.blue2Position, compareTo.myPosition);
score= score + (Math.abs(t1-t2)*weights[SLOPE_TO_BLUE1_WEIGHT]);

// compare distance between myPosition and blue2Position
myDistance = Math.abs(myPosition.row-blue2Position.row) +
   Math.abs(myPosition.column-blue2Position.column);
compareDistance = Math.abs(compareTo.myPosition.row -
                compareTo.blue2Position.row) + Math.abs(
                compareTo.myPosition.column - compareTo.blue2Position.column);
```

```
		difference = Math.abs(myDistance-compareDistance);
		score = score + (Math.abs(difference)*weights[DISTANCE_TO_BLUE2_WEIGHT]);


		// blue3 position
		temp = this.blue3Position.difference(compareTo.blue3Position);
		score+= (temp*weights[BLUE3_POSITION_WEIGHT]);
		t1 = getSlope(this.blue3Position, this.myPosition);
		t2 = getSlope(compareTo.blue3Position, compareTo.myPosition);
		score=score + (Math.abs(t1-t2)*weights[SLOPE_TO_BLUE3_WEIGHT]);

		// compare distance between myPosition and blue3Position
		myDistance = Math.abs(myPosition.row-blue3Position.row) +
		   Math.abs(myPosition.column-blue3Position.column);
		compareDistance = Math.abs(compareTo.myPosition.row -
				compareTo.blue3Position.row) + Math.abs(
				compareTo.myPosition.column - compareTo.blue3Position.column);
		difference = Math.abs(myDistance-compareDistance);
		score = score + (Math.abs(difference)*weights[DISTANCE_TO_BLUE3_WEIGHT]);


		// blue3 position
		temp = this.blue4Position.difference(compareTo.blue4Position);
		score+= (temp*weights[BLUE4_POSITION_WEIGHT]);
		t1 = getSlope(this.blue4Position, this.myPosition);
		t2 = getSlope(compareTo.blue4Position, compareTo.myPosition);
		score= score + (Math.abs(t1-t2)*weights[SLOPE_TO_BLUE4_WEIGHT]);

		// compare distance between myPosition and blue4Position
		myDistance = Math.abs(myPosition.row-blue4Position.row) +
		   Math.abs(myPosition.column-blue4Position.column);
		compareDistance = Math.abs(compareTo.myPosition.row -
				compareTo.blue4Position.row) + Math.abs(
				compareTo.myPosition.column - compareTo.blue4Position.column);
		difference = Math.abs(myDistance-compareDistance);
		score = score + (Math.abs(difference)*weights[DISTANCE_TO_BLUE4_WEIGHT]);


		// red position
		if (this.redPositionKnown && compareTo.redPositionKnown)
		{
			temp = this.redPosition.difference(compareTo.redPosition);
			score+= (temp*weights[RED_POSITION_WEIGHT]);

			// this  change in row/change in column with sign
			double thisSlope = getSlope(this.myPosition, this.redPosition);

			// compare delta row between myPosition and RedPosition
			double		compareToSlope		=		getSlope(compareTo.myPosition,
compareTo.redPosition);
```

```
                        difference = Math.abs(thisSlope - compareToSlope);
                            if (difference > Math.PI)
                            {
                            //          difference = difference - Math.PI;
                            }
                            score+=(weights[SLOPE_TO_RED_WEIGHT] * difference);
                            // compare delta column between myPosition and RedPosition


                            // compare distance between myPosition and RedPosition
                            myDistance = Math.abs(myPosition.row-redPosition.row) +
                               Math.abs(myPosition.column- redPosition.column);
                             compareDistance = Math.abs(compareTo.myPosition.row -
                                                compareTo.redPosition.row) + Math.abs(
                                                compareTo.myPosition.column                                    -
compareTo.redPosition.column);
                            difference = Math.abs(myDistance-compareDistance);
                            score+=(weights[DISTANCE_TO_RED_WEIGHT] * difference);

                    }




                    return (int)score;
            }

            public void print(PrintWriter out) throws IOException
            {
                    // write structure to file
                    out.println("********************");

                    out.println("MyState: " + myState.toString());
                    out.println("MyType: " + myType.toString());
                    out.println("Blue1 Position:  " + blue1Position.row + "," + blue1Position.column);
                    out.println("Blue2 Position:  " +  blue2Position.row + "," + blue2Position.column);
                    out.println("Blue3 Position:  " +  blue3Position.row + "," + blue3Position.column);
                    out.println("Blue4 Position:  " +  blue4Position.row + "," + blue4Position.column);
                    out.println("Red Position:   " + redPosition.row + "," + redPosition.column);
                    out.println("Red Position Known:  " + redPositionKnown);




            }

            private Position myPosition(Situation sit)
            {
                    switch (sit.myType)
                    {
                    case BLUE1:
                            return sit.blue1Position;
```

```
        case BLUE2:
            return sit.blue2Position;
        case BLUE3:
            return sit.blue3Position;
        case BLUE4:
            return sit.blue4Position;
        default:
            return null;
        }
}


private WeightClass getWeights(States context)
{
        double[] weights = new double[20];
        for (int i=0; i< weights.length; i++)
        {
            weights[i]=0;
        }

        // weights[0] - myType

        //weights[1] - blue 1 position
        if (context == States.SEARCH_TOP_LEFT
            )
        {
            weights[BLUE1_POSITION_WEIGHT] =1;
        }
        else
        {
            weights[BLUE1_POSITION_WEIGHT] =0.25;
        }
        // weights[2] - blue 2 position
        if (context == States.SEARCH_TOP_RIGHT
                )
        {
            weights[BLUE2_POSITION_WEIGHT] =1;
        }
        else
        {
            weights[BLUE2_POSITION_WEIGHT] = 0.25;
        }
        // weights[3] - blue 3 position

        if (context == States.SEARCH_BOT_LEFT
                )
        {
            weights[BLUE3_POSITION_WEIGHT] =1;
        }
        else
        {
            weights[BLUE3_POSITION_WEIGHT] = 0.25;
        }

        // weights[4] - blue 4 position

        if (context == States.SEARCH_BOT_RIGHT )
        {
            weights[BLUE4_POSITION_WEIGHT] =1;
        }
        else
        {
            weights[BLUE4_POSITION_WEIGHT] = 0.25;
        }




        weights[RED_POSITION_KNOWN_WEIGHT] =100;


        // weights[5] - redPosition



        // weights[7] - slope to red
        if (context == States.GO_TO_BOTTOM || context == States.GO_TO_LEFT
            || context == States.GO_TO_RIGHT || context == States.GO_TO_TOP)
        {
            weights[SLOPE_TO_RED_WEIGHT] = 3  ;
            weights[DISTANCE_TO_RED_WEIGHT]=1;
            weights[RED_POSITION_WEIGHT] =0;
            weights[MY_POSITION_WEIGHT] = 0;
            weights[BLUE1_POSITION_WEIGHT]= 0;
            weights[BLUE2_POSITION_WEIGHT] = 0;
            weights[BLUE3_POSITION_WEIGHT] = 0;
            weights[BLUE4_POSITION_WEIGHT] = 0;
            weights[SLOPE_TO_BLUE1_WEIGHT] =1.5;
            weights[SLOPE_TO_BLUE2_WEIGHT] =1.5;
            weights[SLOPE_TO_BLUE3_WEIGHT] =1.5;
            weights[SLOPE_TO_BLUE4_WEIGHT ] =1.5;
            weights[DISTANCE_TO_BLUE1_WEIGHT] = 0.2;
            weights[DISTANCE_TO_BLUE2_WEIGHT] = 0.2;
            weights[DISTANCE_TO_BLUE3_WEIGHT] = 0.2;
            weights[DISTANCE_TO_BLUE4_WEIGHT] = 0.2;

        }
        else if (context == States.INTERCEPT_BOTTOM || context ==
States.INTERCEPT_LEFT
            || context == States.INTERCEPT_RIGHT || context ==
States.INTERCEPT_TOP)
        {
                weights[SLOPE_TO_RED_WEIGHT] = 3;
                weights[DISTANCE_TO_RED_WEIGHT]=1;
                weights[RED_POSITION_WEIGHT] =0;
                weights[MY_POSITION_WEIGHT] = 0;
                weights[BLUE1_POSITION_WEIGHT]= 0;
                weights[BLUE2_POSITION_WEIGHT] = 0;
                weights[BLUE3_POSITION_WEIGHT] = 0;
                weights[BLUE4_POSITION_WEIGHT] = 0;
                weights[SLOPE_TO_BLUE1_WEIGHT] =0;
                weights[SLOPE_TO_BLUE2_WEIGHT] =0;
                weights[SLOPE_TO_BLUE3_WEIGHT] =0;
                weights[SLOPE_TO_BLUE4_WEIGHT ] =0;
        }
        else
        {
                weights[SLOPE_TO_RED_WEIGHT] =0;
                weights[DISTANCE_TO_RED_WEIGHT]=0;
                weights[RED_POSITION_WEIGHT] =0;
                weights[MY_POSITION_WEIGHT] = 2;
                weights[SLOPE_TO_BLUE1_WEIGHT] =0;
                weights[SLOPE_TO_BLUE2_WEIGHT] =0;
                weights[SLOPE_TO_BLUE3_WEIGHT] =0;
                weights[SLOPE_TO_BLUE4_WEIGHT ] =0;
        }






        WeightClass wts = new WeightClass();
        wts.weights= weights;
        return wts;

}

// name is deceptive- returns angle from -pi to pi
private double getSlope(Position position1, Position position2)
{
        double slope = 0;
        double temp = (double)(position1.column - position2.column);
        double temp2 = (double)(position1.row - position2.row);
        slope = Math.atan2(temp2, temp );
        return slope;
}

private class WeightClass
{
        double[] weights;

}
```

```
}
```

Constants.java
```
package Common;
// Pursuit game simulation- Cynthia Johnson

public interface Constants
{
    public enum PlayerType  {BLUE1, BLUE2, BLUE3, BLUE4, RED};
    public int NUM_BLUE_PLAYERS = 4;
    public enum Actions {UP, DOWN, RIGHT, LEFT, NONE, NOTIFY_RED_POSITION,
      TRANS_SEARCH_TOP_LEFT, TRANS_SEARCH_TOP_RIGHT, TRANS_SEARCH_BOT_RIGHT,
      TRANS_SEARCH_BOT_LEFT,        TRANS_GO_TO_TOP,        TRANS_GO_TO_RIGHT,
TRANS_GO_TO_LEFT,
      TRANS_GO_TO_BOTTOM,     TRANS_INTERCEPT_TOP,     TRANS_INTERCEPT_BOTTOM,
TRANS_INTERCEPT_RIGHT,
      TRANS_INTERCEPT_LEFT};
    public int EDGE_HIGH = 29;
    public int EDGE_LOW = 0;
    public enum States  {SEARCH_TOP_LEFT, SEARCH_TOP_RIGHT, SEARCH_BOT_LEFT,
      SEARCH_BOT_RIGHT, GO_TO_TOP, GO_TO_RIGHT, GO_TO_LEFT, GO_TO_BOTTOM,
      INTERCEPT_TOP,             INTERCEPT_BOTTOM,             INTERCEPT_RIGHT,
INTERCEPT_LEFT,UNKNOWN};
      public  String[]  stateStrings=   {"search_top_left","search_top_right",
"search_bot_left","search_bot_right","go_to_top",
                  "go_to_right","go_to_left","go_to_bottom", "intercept_top",
                  "intercept_bottom",                        "intecept_right",
"intercept_left","unknown"};

      public enum TeamStates {SEARCHING, BOXING};
}
```

Position.java
```
package Common;
import java.io.Serializable;

public class Position  implements Serializable
{
    public int row;
    public int column;

    public Position()
    {
        row =0;
        column = 0;
    }

    public Position(Position copyPosition)
    {
        row = copyPosition.row;
        column = copyPosition.column;
    }

    public boolean equals(Position newPosition)
    {
        boolean returnValue = false;
        if (newPosition == null)
        {
            return false;
        }
        if (row == newPosition.row && column == newPosition.column)
        {
            returnValue = true;
        }
        return returnValue;
    }

    public int difference(Position otherPosition)
    {
      int difference = 0;

      // get absolute value of difference in row
      difference += Math.abs(row-otherPosition.row);
        // add on absolute value of column difference
      difference += Math.abs(column- otherPosition.column);

      return difference;
    }


}
```

APPENDIX D – PURSUIT GAME RUN-TIME AGENT CODE

Agent.java

```
import Common.Situation;
import Common.Position;



public class Agent extends Player implements Common.Constants
{


        protected Context context;
        protected TeamContext teamContext;
        protected Situation currentSituation;

        protected PlayerType myType;
        protected boolean redPositionKnown;
        protected Position redPosition;



        public Agent(PlayerType type, States startingContext, Position myPos)
        {
                super(myPos.row, myPos.column);
                //get TeamContext
                myType = type;
                teamContext = TeamContext.getInstance();
                context = teamContext.translateNameToContext(startingContext);
                redPositionKnown = false;
        }

  public void move()
  {
        // get my context from teamContext

        // get situation setup
        currentSituation = teamContext.getAgentInfo(myType);
        currentSituation.myPosition = position;
        context = teamContext.translateNameToContext(currentSituation.myState);
         SensorResult result = PursuitGame.getInstance()
                .sensorSweep(position);

      if (result != null)
      {
  // System.out.println(myType.toString() +" Detected red");
        redPositionKnown = true;
        redPosition = result.position;
        sendMessage();
```

```
     }
    PursuitGame.getInstance().setRedKnown(myType, redPositionKnown);
        currentSituation.redPositionKnown = redPositionKnown;
        currentSituation.redPosition = redPosition;
        // now call current context to determine action
        context.step(this,  currentSituation);
        // clear any messages processed this go around

        //redPositionKnown = false;
}

public void receiveMessage(Message message)
{
//      System.out.println("Received message-" + myType.toString());

        if (message.player == PlayerType.RED)
        {
                redPositionKnown = true;
                redPosition= message.position;

        }
}




void setNewContext(States newContext)
{
        context = teamContext.translateNameToContext(newContext);
        teamContext.setNewContext(newContext,myType);

}

public void sendMessage()
{
        if (redPositionKnown)
        {
                PursuitGame.getInstance().sendMessage(PlayerType.RED,
                                redPosition);
        }

}

public Position getCurrentPosition()
{
        return position;
}

public void setCurrentPosition(Position newPosition)
{
      position = newPosition;
    }

    public PlayerType getType()
    {
         return myType;
    }

}
```

Context.java

```
import java.io.EOFException;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.util.Collection;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Set;
import java.util.Random;
import java.util.Vector;


import Common.Constants;
import Common.Situation;
import Common.Position;
import Common.Constants.States;


public  class Context implements Common.Constants
{
        protected String name;
        protected HashMap<Situation, Constants.Actions[]>   policy;


        public Context(String contextName)
        {
                name= contextName;
                initializePolicy(name+".map");
        }


        public void step(Agent owningAgent, Situation currentSituation)
        {

                // check Hashmap for current situation

                        // find nearest neighbor

                        /*
                        #get Collection of keys contained in HashMap using
                        #  Collection values() method of HashMap class
                        # */

                /*      if (this.name.equals(stateStrings[States.SEARCH_BOT_LEFT.ordinal()]))
                        {
                        System.out.println("In go to left");
                        }*/
                         Set set= policy.keySet () ;

                        //obtain an Iterator for Collection
                        Iterator itr = set.iterator();

                        //iterate through HashMap values iterator
                        Actions[] actions = null;
        double lowScore = 1000;
        Situation lowScorer = null;

        int lowScorerCount = 0;
        int count = 0;

         Vector potentialActions = new Vector();
         int matchCount = 0;
         int misMatchCount = 0;
                        while(itr.hasNext())
                        {
                                Situation mySituation = (Situation)itr.next();


                                double                          score                           =
currentSituation.contextuallyWeightedNearestNeighbor(mySituation, currentSituation.myState);
                           if (score == lowScore)
                           {

                                Actions[] actionCheck = (Actions[])policy.get(mySituation);
                                if (!actionsSame(actionCheck, actions))
                                {
                                        boolean inVector = false;
                                   for (int l=0; l< potentialActions.size();l++)
                                   {
                                         ActionCount entry = (ActionCount)potentialActions.get(l);
                                         if (actionsSame(actionCheck, entry.actions))
                                         {
                                                potentialActions.remove(entry);
                                                entry.count ++;
                                                potentialActions.add(entry);
                                                inVector = true;
                                                break;
                                         }
                                   }
                                   if (!inVector)
                                   {
                                        ActionCount entry = new ActionCount();
                                        entry.actions = actionCheck;
                                        entry.count = 1;
                                        potentialActions.add(entry);
                                   }
                                        misMatchCount ++;

                                }
                                else
```

```
            {
                        matchCount ++;
            }

    }
            if (score < lowScore)
            {
                        lowScore = score;

                        lowScorer= mySituation;
                        actions = (Actions[])policy.get(lowScorer);
                        lowScorerCount = count;
                        matchCount = 0;
                        misMatchCount =0;
                        potentialActions = new Vector();

            }
            count ++;
}
int compareCount =matchCount;
Actions[] chosenActions  = actions;
if (misMatchCount > 0 && matchCount < misMatchCount)
{

            // find most prominant action
            for (int l=0; l<potentialActions.size();l++)
            {
                        ActionCount entry = (ActionCount)potentialActions.get(l);
                        if (entry.count >compareCount)
                        {
                                    chosenActions = entry.actions;
                                    compareCount = entry.count;
            //                      System.out.println("Replaced actions");
                        }
            }
}

// set anything necessary in calling Agent
boolean performed = false;
boolean badAction = false;
if (chosenActions != null)
{
            for (int i=0; i<chosenActions.length;i++)
            {

                        Actions action = chosenActions[i];

                        performed = performAction(action,  owningAgent);
                        if (!performed)
                        {
                                    badAction = true;
```

```
                                        //          System.out.println("Bad       Action       for       "       +
owningAgent.myType.toString());
                                                    break;
                                        }

                              }
                    }

          }



     private boolean performAction(Actions action,  Agent owningAgent)
     {
          boolean actionPerformed = true;
          // get current Agent position
       Position position = new Position();
       position.row = owningAgent.getCurrentPosition().row;
       position.column = owningAgent.getCurrentPosition().column;
          switch (action)
          {
          case UP:
               // move Agent up
               position.row--;

               break;

          case DOWN:
               // move agent down
               position.row++;
               break;

          case LEFT:
               // move agent left
               position.column--;
               break;

          case RIGHT:
               // move agent right
               position.column++;
               break;

          case NOTIFY_RED_POSITION:
               owningAgent.sendMessage();
               break;
          case TRANS_SEARCH_TOP_LEFT:
               TeamContext.getInstance().setNewContext(States.SEARCH_TOP_LEFT,
                         owningAgent.getType());
               break;

          case TRANS_SEARCH_TOP_RIGHT:
```

```
          TeamContext.getInstance().setNewContext(States.SEARCH_TOP_RIGHT,
                              owningAgent.getType());
          break;
case TRANS_SEARCH_BOT_RIGHT:
          TeamContext.getInstance().setNewContext(States.SEARCH_BOT_RIGHT,
                              owningAgent.getType());
          break;

case TRANS_SEARCH_BOT_LEFT:
          TeamContext.getInstance().setNewContext(States.SEARCH_BOT_LEFT,
                              owningAgent.getType());
          break;

case TRANS_GO_TO_TOP:
          TeamContext.getInstance().setNewContext(States.GO_TO_TOP,
                              owningAgent.getType());
          break;

case TRANS_GO_TO_RIGHT:
          TeamContext.getInstance().setNewContext(States.GO_TO_RIGHT,
                              owningAgent.getType());
          break;

case TRANS_GO_TO_LEFT:
          TeamContext.getInstance().setNewContext(States.GO_TO_LEFT,
                              owningAgent.getType());
          break;

case TRANS_GO_TO_BOTTOM          :
          TeamContext.getInstance().setNewContext(States.GO_TO_BOTTOM,
                              owningAgent.getType());
          break;

case TRANS_INTERCEPT_TOP :
          TeamContext.getInstance().setNewContext(States.INTERCEPT_TOP,
                              owningAgent.getType());
          break;

case TRANS_INTERCEPT_BOTTOM :
          TeamContext.getInstance().setNewContext(States.INTERCEPT_BOTTOM,
                              owningAgent.getType());
          break;

case TRANS_INTERCEPT_LEFT :
          TeamContext.getInstance().setNewContext(States.INTERCEPT_LEFT,
                              owningAgent.getType());
          break;

case TRANS_INTERCEPT_RIGHT :
          TeamContext.getInstance().setNewContext(States.INTERCEPT_RIGHT,
                              owningAgent.getType());
```

```
                        break;


    default:
              // do nothing
             break;
            }
            boolean happens=PursuitGame.getInstance().moveTo(owningAgent.getType()
                                , position);
            if (happens)
            {
                    owningAgent.setCurrentPosition(position);
            }
            else if (action == Actions.UP || action == Actions.DOWN ||
                                action == Actions.LEFT || action == Actions.RIGHT)
            {
                    actionPerformed = false;
            }
            return actionPerformed;

}



private void initializePolicy(String fileName)
{
            ObjectInputStream input = null;
            try // open file
      {
         input = new ObjectInputStream(
            new FileInputStream( fileName ) );
      } // end try
      catch ( IOException ioException )
      {
         System.err.println( "Error opening file." );
      } // end catch

              // read in single HashMap of policy generated by
      // learning algorithm
      try // input the values from the file
      {
               Object tempPolicy = input.readObject();
               if (tempPolicy instanceof HashMap<?,?>)
               {
                    policy = (HashMap<Situation, Constants.Actions[]>)tempPolicy;
               }


      } // end try
      catch ( EOFException endOfFileException )
      {
         System.out.println("end of file reached- nothing read");
        } // end catch
        catch ( ClassNotFoundException classNotFoundException )
        {
                classNotFoundException.printStackTrace();
          System.err.println( "Unable to create object." );
        } // end catch
        catch ( IOException ioException )
        {
                ioException.printStackTrace();
          System.err.println( "Error during reading from file." );
        } // end catch

    }

    //method to check if actions are equal
    private boolean actionsSame(Actions[] compareTo, Actions[] compareAgainst)
    {
        boolean same = false;
        if (compareTo!= null && compareAgainst != null)
        {
            if (compareTo.length == compareAgainst.length)
            {
                same = true;
                for (int i=0;i<compareTo.length;i++)
                {
                    if (compareTo[i]!=compareAgainst[i]){
                        same = false;
                    }
                }
            }
        }

        return same;
    }

    private class ActionCount
    {
        public Actions[] actions = null;
        public int count = 0;
    }
}
```

```
TeamContext.java
import Common.Constants;
import Common.Situation;
import Common.Position;

// team context for CCXBR framework
// author: Cynthia Johnson
public class TeamContext implements Constants
{
	private static boolean initialized = false;
	private static TeamContext instance = null;


	// initialize all the contexts in this game
	private Context searchTopLeft= new Context(stateStrings[States.SEARCH_TOP_LEFT.ordinal()]);
	private Context searchTopRight = new Context(stateStrings[States.SEARCH_TOP_RIGHT.ordinal()]);
	private Context searchBotLeft = new Context(stateStrings[States.SEARCH_BOT_LEFT.ordinal()]);
	private Context searchBotRight= new Context(stateStrings[States.SEARCH_BOT_RIGHT.ordinal()]);
	private Context goToTop = new Context(stateStrings[States.GO_TO_TOP.ordinal()]);
	private Context goToRight = new Context(stateStrings[States.GO_TO_RIGHT.ordinal()]);
	private Context goToLeft = new Context(stateStrings[States.GO_TO_LEFT.ordinal()]);
	private Context goToBottom = new Context(stateStrings[States.GO_TO_BOTTOM.ordinal()]);
	private Context interceptTop = new Context(stateStrings[States.INTERCEPT_TOP.ordinal()]);
	private Context interceptBottom = new Context(stateStrings[States.INTERCEPT_BOTTOM.ordinal()]);
	private Context interceptRight = new Context(stateStrings[States.INTERCEPT_RIGHT.ordinal()]);
	private Context interceptLeft = new Context(stateStrings[States.INTERCEPT_LEFT.ordinal()]);
	private TeamStates teamContext = TeamStates.SEARCHING;
	States blues[] = new States[4];

	private TeamContext()
	{
		blues[PlayerType.BLUE1.ordinal()] = States.SEARCH_TOP_LEFT;
		blues[PlayerType.BLUE2.ordinal()] = States.SEARCH_TOP_RIGHT;
		blues[PlayerType.BLUE3.ordinal()] = States.SEARCH_BOT_LEFT;
		blues[PlayerType.BLUE4.ordinal()] = States.SEARCH_BOT_RIGHT;

	}

	public static TeamContext getInstance()
	{
		// method to allow agents to get instance of team context
		if (!initialized)
		{
			instance = new TeamContext();
			initialized = true;
```

```
        }

        return instance;
}

public void destroy()
{
        instance = null;
        initialized = false;
}

public Situation getAgentInfo(PlayerType type)
{
        // returns the situation for the agent at
        // node numbered node.
        Situation situation = new Situation();
        // fill information in the situation
        situation.myType = type;
        situation.myState = blues[type.ordinal()];
        // they know all blue and red only if known
   situation.blue1Position = PursuitGame.getInstance().
    getPosition(PlayerType.BLUE1);
   situation.blue2Position = PursuitGame.getInstance().
    getPosition(PlayerType.BLUE2);
   situation.blue3Position = PursuitGame.getInstance().
    getPosition(PlayerType.BLUE3);
   situation.blue4Position = PursuitGame.getInstance().
    getPosition(PlayerType.BLUE4);
   situation.redPosition = PursuitGame.getInstance().
    getPosition(PlayerType.RED);


        return situation;
}

Context translateNameToContext(States contextName)
{

        switch (contextName)
        {
        case SEARCH_TOP_LEFT:
                return searchTopLeft;

        case SEARCH_TOP_RIGHT:
                return searchTopRight;

        case SEARCH_BOT_LEFT:
                return searchBotLeft;

        case SEARCH_BOT_RIGHT:
                return searchBotRight;
        case GO_TO_TOP:
                return goToTop;

        case GO_TO_RIGHT:
                return goToRight;

        case GO_TO_LEFT:
                return goToLeft;

        case GO_TO_BOTTOM:
                return goToBottom;

        case INTERCEPT_TOP:
                return interceptTop;

        case INTERCEPT_BOTTOM:
                return interceptBottom;

        case  INTERCEPT_RIGHT:
                return interceptRight;

        case INTERCEPT_LEFT:
                return interceptLeft;

        default:
                return null;
        }
}

public States getContext(int index)
{
        return blues[index];
}

public void setNewContext(States newContext, PlayerType type)
{
//      System.out.println("Switching context "+ type.toString()+" to " +
//                      newContext.toString());
        // set context of that player
        switch (type)
        {
        case BLUE1:
                blues[0]=newContext;
                break;
        case BLUE2:
                blues[1] = newContext;
                break;
        case BLUE3:
                blues[2] = newContext;
                break;
                case BLUE4:
                        blues[3] = newContext;
                        break;
                }

        }


}
```

APPENDIX E - TEAMBOTS TRAINER CLASS CODE

Trainer.java

```
package Training;
// main class for training algorithm for bucket brigade
import Common.Constants;
import Common.Situation;
import Common.Position;
import Common.PlayerInfo;


import java.io.BufferedReader;
import java.io.EOFException;
import java.io.FileReader;
import java.io.FileOutputStream;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;

import java.io.ObjectOutputStream;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Iterator;
import java.util.List;
import java.util.Map;
import java.util.Set;
import java.util.Vector;

import EDU.gatech.cc.is.util.Vec2;

public class Trainer implements Constants
{
        private BufferedReader in;
        private String logName = null;
        private     List<Map<Situation,     Constants.ACTION_PRIMITIVE>>     maps     =     new
ArrayList<Map<Situation, Constants.ACTION_PRIMITIVE>>();
        private Vector[] trainingData = new Vector[14];
   private final int TRANS_GO_TO_DISTANCE = 5;

   private Vec2 opponentGoal = new Vec2(-1.37,0);
   private Vec2 goal = new Vec2(1.37,0);

        public Trainer(String fileName)
        {

            logName = fileName;
                try
                {
                // open file for reading
                        in= new BufferedReader(new FileReader(logName));
                }
                catch (Exception ex)
                {
```

```
				System.out.println("Error opening log file");
				ex.printStackTrace();
			}

			//create a hash map for each context
			for (int i=0; i< Constants.States.END_CONTEXT.ordinal(); i++)
			{
				maps.add(new HashMap<Situation, Constants.ACTION_PRIMITIVE>());
				trainingData[i] = new Vector();
			}

		}

		public void createHashMaps()
		{



			Situation current = new Situation();


			int currentSimTime=0;


			// process 1 agent at a time - so we'll be iterating through
			// the file multiple times
			for (int j=0; j<=5; j++)
			{
				// reset file to beginning
				try
				{
						in.close();
						in= new BufferedReader(new FileReader(logName));
					do
					{


						// looking for agent j
						// create current Situation
						current = new Situation();


						// first line should be sim time
						String timeLine = in.readLine();
						// get sim time from parsing the first line
						String[] splits = timeLine.split("\t");

						int simTime = Integer.parseInt(splits[1]);
//		System.out.println("Processed a simTime " + simTime+ " for agent " + j);
						// we are building situation for west player j
						// other west players go into teammates
// east players go into opponents
int teammateCount = 0;
int opponentCount = 0;
// process all players - ten total for both sides
for (int i=0; i< 10; i++)
{
	int playerNum = 0;
	boolean isWestSide = false;
	Position position = new Position();
	String description = new String("");
	// first line = side Player Number: tab playerNum
	String line = in.readLine();
	splits = line.split("\t");
	// this creates two strings- the first has side info- the
second player Num

	playerNum = Integer.parseInt(splits[1]);
	isWestSide = splits[0].contains("West");

	// next line contains description
	// not used - read and throw away
	line = in.readLine();

	// next line = x, y position info
	line = in.readLine();
	splits = line.split("\t");
	position.x = Double.parseDouble(splits[1]);
	position.y = Double.parseDouble(splits[2]);

	// next line = heading
	line = in.readLine();
	splits = line.split("\t");
	position.t = Double.parseDouble(splits[1]);

	// next line = speed
	line = in.readLine();
	splits = line.split("\t");
	double speed = Double.parseDouble(splits[1]);
	// ignore this- speed not used in determining next move


	// end of player - place appropriately in situation
	if (!isWestSide && playerNum == j)
	{
		current.myNum = playerNum;
		current.myPosition = position;
	}
	else if (!isWestSide)
	{
		PlayerInfo info = new PlayerInfo();

		info.position = position;
```

```
				info.playerNum = playerNum;
				current.teammates[teammateCount] = info;
				teammateCount++;
			}
			else // is opponent
			{
				PlayerInfo info = new PlayerInfo();
				info.position = position;
				info.playerNum = playerNum;
				current.opponents[opponentCount] = info;
				opponentCount++;
			}

		}  // end parse of player Info

		// adjust teammates and opponent to be angle and
		// distance rather than actual postiion
		for (int i=0; i<4; i++)
		{
			current.teammates[i].position.sub(current.myPosition);
		}
		for (int i=0; i<5; i++)
		{
			current.opponents[i].position.sub(current.myPosition);
		}
		// Next is ball position
		Position ballPosition = new Position();
		String line = in.readLine();
		splits = line.split("\t");
		if (splits.length >2 )
		{

			ballPosition.x = Double.parseDouble(splits[1]);
			ballPosition.y = Double.parseDouble(splits[2]);
			current.ballPosition = ballPosition;
		}
		if (ballPosition.x >= 0.0)
		{
			current.teamState = Constants.TeamStates.DEFENSE;
		}
		else
		{
			current.teamState = Constants.TeamStates.OFFENSE;
		}
		// next is ball velocity
		Position ballVelocity = new Position();
		line = in.readLine();
		splits = line.split("\t");
		if (splits.length > 1)
		{
```

```
                    ballPosition.t = Double.parseDouble(splits[1]);;
                    // ball velocity not used to determine next move
                    //current.ballVelocity = ballVelocity;
                }
                // next line is score
                // read and ignore
                line = in.readLine();

                // next line is After:  read and ignore
                line = in.readLine();

                // now get appropriate actions with this scenario from file
                Constants.ACTION_PRIMITIVE          currentActions          =
Constants.ACTION_PRIMITIVE.NONE;
                String contextDescription = new String();
                // we have five players to parse - we only want playerNum =j,
                // but we have to parse all of them to get currentActions

                for (int i=0; i< 5; i++)
                {

                    int playerNum = 0;


                    String description = new String("");
                    // first line = side Player Number: tab playerNum
                  line = in.readLine();
                    splits = line.split("\t");
                    // this creates two strings- the first has side info- the
second player Num
                    playerNum = Integer.parseInt(splits[1]);


                    // next line contains description
                    // used to help with out context designation
                    description = in.readLine();

                    // next line = heading info
                    double heading = 0.0;
                    line = in.readLine();
                    splits = line.split("\t");
                    heading = Double.parseDouble(splits[1]);

                    // next line = speed
                    double speed = 0.0;
                    line = in.readLine();
                    splits = line.split("\t");
                    speed = Double.parseDouble(splits[1]);

                    // next line = kicked true or false
                    boolean kicked = false;
```

```
					line = in.readLine();
					splits = line.split("\t");
					kicked = Boolean.parseBoolean(splits[1]);

					if (playerNum == j)
					{
						// this is our action info
//						currentActions.speed = speed;
//						currentActions.kick = kicked;
//						currentActions.heading = heading;
						//contextDescription = description;
						if (description.equals("move to ball"))
						{
							currentActions					=
ACTION_PRIMITIVE.MOVE_TO_BALL;
						}
						else if (description.equals("get behind ball"))
						{
							currentActions					=
ACTION_PRIMITIVE.GET_BEHIND_BALL;

						}
						else if (description.equals("move to backfield"))
						{
							currentActions					=
ACTION_PRIMITIVE.MOVE_TO_BACKFIELD;
						}
						else
						{
							System.out.println("Unknonw action= "
+ description);
						}
					}
				}

				// this version only does east side
				current.side = 1;
				// determine if closest to ball
				double				distance				=
current.myPosition.differenceXY(current.ballPosition);
				//current.closestToBall = false;
				// check teammates to see if one of them is closer

				for (int i=0; i<4;i++)
				{
					double				temp				=
current.teammates[i].position.differenceXY(current.ballPosition);

				}
				boolean ballOnOurSide = false;
				if (current.ballPosition.x < 0)
```

```
                        ballOnOurSide = true;




                // calculate ball angle

                double     temp     =     (double)(current.ballPosition.x     -
current.myPosition.x);
                double     temp2     =     (double)(current.ballPosition.y     -
current.myPosition.y);
                current.ballAngle = Math.atan2(temp2, temp );

                // end parse of one simtime
                // default context based on player number
                Constants.States context = Constants.States.MOVING_0 ;
                // is player behind ball - use x position to determine
                if (current.ballPosition.x < current.myPosition.x)
                {
                        current.isBehindBall = true;
                }



                // determine context based on context Description
                // default to to_ball context
                Constants.States myContext = Constants.States.MOVING_0;
                // determine context


                if (current.isBehindBall  )
                {
                        switch (current.myNum)
                        {
                        case 0:
                                myContext = Constants.States.MOVING_0;
                                break;
                        case 1:
                                myContext = Constants.States.MOVING_1;
                                break;
                        case 2:
                                myContext = Constants.States.MOVING_2;

                                break;
                        case 3:
                                myContext = Constants.States.MOVING_3;
                                break;
                        case 4:
                                myContext = Constants.States.MOVING_4;
                                break;
```

```
                                        }
                                }
                                else if (!current.isBehindBall  )
                                {
                                        switch (current.myNum)
                                        {
                                        case 0:
                                                myContext                                       =
Constants.States.GET_BEHIND_0;
                                                break;
                                        case 1:
                                                myContext                                       =
Constants.States.GET_BEHIND_1;
                                                break;
                                        case 2:
                                                myContext                                       =
Constants.States.GET_BEHIND_2;
                                                break;
                                        case 3:
                                                myContext                                       =
Constants.States.GET_BEHIND_3;
                                                break;
                                        case 4:
                                                myContext                                       =
Constants.States.GET_BEHIND_4;
                                                break;
                                        }
                                }
                                current.myState = myContext;
                                context = myContext;

                                // save to behavior map
                                trainHashMap(current, context,  currentActions);

                                // loop until EOF
                        } while (true);

                }   // end try
                catch (EOFException endOfFileException)
                {
                        // done with this agent =
                        // go on to the next one
                        System.out.println("Finished with an agent-EOF");
                }
                catch (Exception ex)
                {
                        //ex.printStackTrace();
                        System.out.println("Finished with an agent");
                }
        }  // end for (j)
```

```
		// done with agents
		// save policies to files
		savePoliciesToFile();


	}


	private void trainHashMap(Situation situation, Constants.States context, 
Constants.ACTION_PRIMITIVE actions)
	{
		HashMap<Situation, Constants.ACTION_PRIMITIVE> myMap = null;

		boolean mismatch = false;

		try
		{

			// sort arrays in Situation
			situation.sortArrays();
		}
		catch (Exception ex)
		{
			//Unable to sort
			ex.printStackTrace();
		}

		// get map based on context
		int index = context.ordinal();


		myMap = (HashMap<Situation,Constants.ACTION_PRIMITIVE>)maps.get(index);

		Set set= myMap.keySet () ;

		//obtain an Iterator for Collection

		Iterator itr = set.iterator();

		//iterate through HashMap values iterator
		boolean found = false;
		while(itr.hasNext())
		{

			Situation latestKey = (Situation)itr.next();
			// check to see if this situation exists in map already
			//if (latestKey.isSimilar(situation))
			if (latestKey.equals(situation))
			{
				found = true;
				//System.out.println("Got a matching situation");
```

```
                                        // check to see if actions map
                                        Constants.ACTION_PRIMITIVE                savedActions                 =
(Constants.ACTION_PRIMITIVE)myMap.get(latestKey);
                                        // first check for size map
                                        if (savedActions == null)
                                        {
                                                System.out.println("Got null saved actions");
                                        }
                                        else if (actions == null)
                                        {
                                                System.out.println("actions = null");

                                        }
                                        else if (!(savedActions == actions) )
                                        {
                                                // different actions- save a separate entity unless equal
                                                if (latestKey.equals(situation))
                                                {

                                                        // mismatch in actions
                                                        mismatch = true;
                                                        System.out.println("Got a mismatched situation");
                                                }
                                                else if (savedActions==actions)
                                                {
                                                        found = true;
                                                        System.out.println("Saved entering one");
                                                }
                                                else
                                                {
                                                        found = false;
                                                }
                                        }
                                        // we have mismatch or duplicate mark appropriately
                                        if (mismatch)
                                        {
                                                // get entry from myVector and increment count
                                                System.out.println("GOT A MISMATCH");

                                        }

                                }

                        }
                        if (!found)
                        {
                                // this is first time for this situation so just add to map
                                myMap.put(situation, actions);
                        }

                }
```

```
// method to convert string to boolean
boolean convertStringToBoolean(String boolString)
{
        if (boolString.equals("false"))
        {
                return false;
        }
        else
        {
                return true;
        }
}





private void savePoliciesToFile()
{
        for (int k=0; k< Constants.States.END_CONTEXT.ordinal(); k++)
        {
                ObjectOutputStream output = null;
                String fileName = "default.map";
                // get policy hashmap to save
                HashMap<Situation, Constants.ACTION_PRIMITIVE> myMap = null;

                // get map based on context
                myMap = (HashMap<Situation, Constants.ACTION_PRIMITIVE>)maps.get(k);
                // get name of file to save to
                String temp = stateStrings[k];
                fileName = temp+".map";

                // open file for writing
                try // open file
              {
                        output = new ObjectOutputStream(
                          new FileOutputStream( fileName ) );

                        // write to file
                        output.writeObject( myMap );

                        // close file
                        output.close();
              } // end try

                catch (IOException ioException)
```

```
            {
                 ioException.printStackTrace();
              System.err.println( "Error opening file." );
            } // end catch

               // open file for writing
               try // open file
            {
                         String txtFileName = temp + ".txt";
                      PrintWriter out = new PrintWriter(
                        new FileWriter( txtFileName ) );

                      Set keySet = myMap.keySet();
                      Iterator iter = keySet.iterator();
                      while (iter.hasNext())
                      {
                          Situation sit = (Situation)iter.next();
                          sit.print(out);
                          Constants.ACTION_PRIMITIVE actions = myMap.get(sit);
                          out.println("Action =" + actions.toString());
                         // out.println("Heading = " + actions.heading);
                         // out.println("Speed = " + actions.speed);
                         // out.println("Kick = " + actions.kick);


                      }


                      // close file
                      out.close();
            } // end try

               catch (IOException ioException)
            {
                 ioException.printStackTrace();
              System.err.println( "Error opening file." );
            } // end catch
      } // end for k
}

      /**
       * @param args
       */
      public static void main(String[] args)
      {
               // get log file name with training data
               // from arguments or constant
               String logFileName = "TeamBotsTrain.log";
               // create trainer class
               Trainer myTrainer = new Trainer(logFileName);
```

```
                myTrainer.createHashMaps();



        }

}
```

Situation.java
```
package Common;

import  EDU.gatech.cc.is.util.Vec2;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.io.Serializable;
import java.util.Arrays;
import java.util.Comparator;


// Class describing the current world state of an agent

//Class describing the current world state of an agent

public class Situation implements Constants, Serializable
{

   public int  myNum;
   public Constants.States myState;
          public Position myPosition = new Position();
          public PlayerInfo teammates[] = new PlayerInfo[4];
          public PlayerInfo opponents[] = new PlayerInfo[5];
          public Position ballPosition = new Position();
          public double ballAngle ;


          public boolean isBehindBall;
          public int side;   // = -1 if west and 1 if east
          public Constants.TeamStates teamState;
          // set up constants for various weights
          final int MY_POSITION_WEIGHT =    0;
          final int TEAMMATE1_WEIGHT =      1;
          final int TEAMMATE2_WEIGHT =      2;
          final int TEAMMATE3_WEIGHT =      3;
          final int TEAMMATE4_WEIGHT =      4;
          final int OPPONENT1_WEIGHT =      5;
          final int OPPONENT2_WEIGHT =      6;
          final int OPPONENT3_WEIGHT =      7;
          final int OPPONENT4_WEIGHT =      8;
          final int OPPONENT5_WEIGHT =      9;
          final int BALL_WEIGHT =           10;
          final int SIDE_WEIGHT =                     11;
          final int TEAMSTATE_WEIGHT =    12;
   final int BEHIND_BALL_WEIGHT =           13;
   final int BALL_ANGLE_WEIGHT =   14;
          final int PLAYER_NUM_WEIGHT  =   15;
          final double SIMILAR =                      0.50;
```

```
// default constructor
public Situation()
{

            isBehindBall = false;
            for (int i=0; i< 4; i++)
            {
                        teammates[i] = new PlayerInfo();
                        opponents[i] = new PlayerInfo();


            }
            opponents[4] = new PlayerInfo();
            teamState = Constants.TeamStates.DEFENSE;

}

// copy constructor
public Situation(Situation copy)
{

myNum = copy.myNum;
            myPosition = copy.myPosition;
            for (int i =0 ; i < 4; i++)
            {
                        teammates[i] = copy.teammates[i];
            }
   for (int i=0; i<5; i++)
   {
            opponents[i] = copy.opponents[i];
   }

            ballPosition = copy.ballPosition;

            isBehindBall = copy.isBehindBall;
            teamState = copy.teamState;

}


public boolean equals(Situation copy)
{
            // determine if newSituation is equal to this one
            boolean result = false;
            if (
myNum == copy.myNum &&
            myPosition == copy.myPosition &&
            teammates[0] == copy.teammates[0] &&
            teammates[1] == copy.teammates[1] &&
            teammates[2] == copy.teammates[2] &&
            teammates[3] == copy.teammates[3] &&
        opponents[0] == copy.opponents[0] &&
        opponents[1] == copy.opponents[1] &&
        opponents[2] == copy.opponents[2] &&
        opponents[3] == copy.opponents[3] &&
        opponents[4] == copy.opponents[4] &&
        ballPosition == copy.ballPosition &&
        isBehindBall == copy.isBehindBall  &&
        teamState    == copy.teamState           )
        {
                result = true;
        }

        return result;
}



public double contextuallyWeightedNearestNeighbor(Situation compareTo, States context)
{
        WeightClass weightClass = getWeights(context);
        double weights[] = weightClass.weights;
        double score = 0;

        // make sure positions are appropriately sorted for teammates and opponents
        this.sortArrays();
        compareTo.sortArrays();

        // my position difference
        score = myPosition.differenceXY(compareTo.myPosition) *
                weights[MY_POSITION_WEIGHT];

        // teammate differences
        score+= teammates[0].overallDifference(compareTo.teammates[0]) *
                weights[TEAMMATE1_WEIGHT];

        score+= teammates[1].overallDifference(compareTo.teammates[1]) *
                weights[TEAMMATE2_WEIGHT];

        score+= teammates[2].overallDifference(compareTo.teammates[2]) *
                weights[TEAMMATE3_WEIGHT];

        score+= teammates[3].overallDifference(compareTo.teammates[3]) *
                weights[TEAMMATE4_WEIGHT];



        // opponent differences
        score+= opponents[0].overallDifference(compareTo.opponents[0]) *
                weights[OPPONENT1_WEIGHT];

        score+= opponents[1].overallDifference(compareTo.opponents[1]) *
```

```
				weights[OPPONENT2_WEIGHT];

		score+= opponents[2].overallDifference(compareTo.opponents[2]) *
				weights[OPPONENT3_WEIGHT];

		score+= opponents[3].overallDifference(compareTo.opponents[3]) *
				weights[OPPONENT4_WEIGHT];

		score+= opponents[4].overallDifference(compareTo.opponents[4]) *
				weights[OPPONENT5_WEIGHT];

		// ball position
		score += ballPosition.overallDifference(compareTo.ballPosition) *
				weights[BALL_WEIGHT];

		score += Math.abs(ballAngle - compareTo.ballAngle) *
				weights[BALL_ANGLE_WEIGHT];



		// side different
		score+= Math.abs(side - compareTo.side) *
				weights[SIDE_WEIGHT];

		// team context difference
		score += Math.abs(teamState.ordinal() - compareTo.teamState.ordinal()) *
				weights[TEAMSTATE_WEIGHT];

		if (myNum != compareTo.myNum)
		{
				score +=weights[PLAYER_NUM_WEIGHT];
		}
		if (isBehindBall != compareTo.isBehindBall)
		{
				score +=weights[BEHIND_BALL_WEIGHT];
		}

		return (double)score;
	}

	public boolean isSimilar(Situation other)
	{
		double score = contextuallyWeightedNearestNeighbor(other, States.END_CONTEXT);
		// call nearest neighbor with context to return all ones
		if ( score < SIMILAR)
		{
		//		System.out.println("Score is " + score);
				return true;
		}
		return false;
	}
```

```
public void sortArrays()
{
            // sort teammates and opponents using Position as comparator
            Arrays.sort(teammates, (Comparator)opponents[0]);
            Arrays.sort(opponents, (Comparator)teammates[0]);
}


public void print(PrintWriter out) throws IOException
{
            // write structure to file
            out.println("********************");




            out.println("MyNum: " + myNum);
    // TBD print position information
    out.println("My Position: " + myPosition.x +"," + myPosition.y);
    out.println("Ball Position: " + ballPosition.x + "," + ballPosition.y);
    out.println("My heading: " + myPosition.t);
    out.println("Ball heading " + ballPosition.t);
    out.println("Ball Angle " + ballAngle);
    out.println("Is Behind Ball " + isBehindBall);


}




private WeightClass getWeights(States context)
{
            double[] weights = new double[25];
            for (int i=0; i< weights.length; i++)
            {
                        weights[i]=0;
            }

            if (context == Constants.States.MOVING_0)
            {
                        weights[MY_POSITION_WEIGHT] = 1.0          ;
                        weights[TEAMMATE1_WEIGHT] = 0.0;
                        weights[TEAMMATE2_WEIGHT] = 0.0;
                        weights[TEAMMATE3_WEIGHT] = 0.0;
                        weights[TEAMMATE4_WEIGHT] =0.0;
                        weights[OPPONENT1_WEIGHT] =0.0;
                        weights[OPPONENT2_WEIGHT] =00.0;
                        weights[OPPONENT3_WEIGHT] = 0.0;
                        weights[OPPONENT4_WEIGHT] =0.0;
                        weights[OPPONENT5_WEIGHT] =0.0;
                        weights[BALL_WEIGHT] =1.0;
                        weights[SIDE_WEIGHT] =0.0;
                        weights[TEAMSTATE_WEIGHT]= 0.0;
                        weights[BEHIND_BALL_WEIGHT] = 0.0;
                        weights[BALL_ANGLE_WEIGHT] = 2;
                        weights[PLAYER_NUM_WEIGHT] = 1.0;
            }
            else if(context == Constants.States.MOVING_1  ||
                                    context == Constants.States.MOVING_3 ||
                                    context == Constants.States.MOVING_2 ||
                                    context == Constants.States.MOVING_4)
            {
                        weights[MY_POSITION_WEIGHT] = 1.0         ;
                        weights[TEAMMATE1_WEIGHT] = 0.1;
                        weights[TEAMMATE2_WEIGHT] = 0.1;
                        weights[TEAMMATE3_WEIGHT] = 0.1;
                        weights[TEAMMATE4_WEIGHT] =0.1;
                        weights[OPPONENT1_WEIGHT] =0.0;
                        weights[OPPONENT2_WEIGHT] =00.0;
                        weights[OPPONENT3_WEIGHT] = 0.0;
                        weights[OPPONENT4_WEIGHT] =0.0;
                        weights[OPPONENT5_WEIGHT] =0.0;

                        weights[BALL_WEIGHT] =0.5;
                        weights[SIDE_WEIGHT] =0.0;
                        weights[TEAMSTATE_WEIGHT]= 0.0;
                        weights[PLAYER_NUM_WEIGHT] = 1.0;
                        weights[BEHIND_BALL_WEIGHT] = 1.0;
                        weights[BALL_ANGLE_WEIGHT] = 1.0;
            }
            else if (context == Constants.States.GET_BEHIND_0 ||
                                     context == Constants.States.GET_BEHIND_1 ||
                                     context == Constants.States.GET_BEHIND_2 ||
                                     context == Constants.States.GET_BEHIND_3 ||
                                     context == Constants.States.GET_BEHIND_4)
            {
                        weights[MY_POSITION_WEIGHT] =1.0;
                        weights[TEAMMATE1_WEIGHT] = 0.1;
                        weights[TEAMMATE2_WEIGHT] = 0.1;
                        weights[TEAMMATE3_WEIGHT] = 00.1;
                        weights[TEAMMATE4_WEIGHT] =0.1;
                        weights[OPPONENT1_WEIGHT] =0.0;
                        weights[OPPONENT2_WEIGHT] =0.0;
                        weights[OPPONENT3_WEIGHT] = 0.0;
                        weights[OPPONENT4_WEIGHT] =0.0;
                        weights[OPPONENT5_WEIGHT] =0.0;
                        weights[BALL_WEIGHT] =1.0;

                        weights[SIDE_WEIGHT] =0;
                        weights[TEAMSTATE_WEIGHT]= 0.0;
                        weights[PLAYER_NUM_WEIGHT] = 1.0;
```

```
                    weights[BEHIND_BALL_WEIGHT] = 0;
                    weights[BALL_ANGLE_WEIGHT] = 2.0;
            }
            else
            {
                    weights[MY_POSITION_WEIGHT] = 1.0;
                    weights[TEAMMATE1_WEIGHT] = 1.0;
                    weights[TEAMMATE2_WEIGHT] = 1.0;
                    weights[TEAMMATE3_WEIGHT] = 1.0;
                    weights[TEAMMATE4_WEIGHT] =1.0;
                    weights[OPPONENT1_WEIGHT] =1.0;
                    weights[OPPONENT2_WEIGHT] =1.0;
                    weights[OPPONENT3_WEIGHT] = 1.0;
                    weights[OPPONENT4_WEIGHT] =1.0;
                    weights[OPPONENT5_WEIGHT] =1.0;
                    weights[BALL_WEIGHT] =1.0;

                    weights[SIDE_WEIGHT] =1.0;
                    weights[TEAMSTATE_WEIGHT]= 1.0;

                    weights[PLAYER_NUM_WEIGHT] = 1.0;
                    weights[BEHIND_BALL_WEIGHT] = 1.0;
                    weights[BALL_ANGLE_WEIGHT] = 1.0;
            }


            WeightClass returnValue = new WeightClass();
            returnValue.weights= weights;
            return returnValue;
    }




    private class WeightClass
    {
            double[] weights;

    }
}
```

Constants.java

```
package Common;
// TeamBot simulation- Cynthia Johnson

public interface Constants
{

   public enum States  {CLOSEST, MOVING_0, MOVING_1, MOVING_2,
         MOVING_3, MOVING_4, GET_BEHIND_0, GET_BEHIND_1,
         GET_BEHIND_2, GET_BEHIND_3,GET_BEHIND_4, END_CONTEXT};
         public String[] stateStrings=  { "CLOSEST", "MOVING_0",
                          "MOVING_1","MOVING_2", "MOVING_3", "MOVING_4",
                          "GET_BEHIND_0", "GET_BEHIND_1", "GET_BEHIND_2",
                          "GET_BEHIND_3","GET_BEHIND_4"};
         public enum ACTION_PRIMITIVE  {GET_BEHIND_BALL, MOVE_TO_BALL,
                   MOVE_TO_BACKFIELD, NONE

         };
         public String[] primitiveStrings = {"get behind ball",
                          "move to ball", "move to backfield"
         };

         public enum TeamStates {OFFENSE, DEFENSE};
}
```

PlayerInfo.java

```
package Common;
/**
 * @(#)PlayerInfo.java
 *
 *
 * @author
 * @version 1.00 2010/11/12
 */
 import java.io.Serializable;
 import java.util.Comparator;

// Data structure class for player info in teammates and opponents
public class PlayerInfo implements Serializable, Comparator
{
	public int playerNum = -1;
	public Position position = new Position();

	public int compare(Object o1, Object o2)
	{
		PlayerInfo player1 = (PlayerInfo)o1;
		PlayerInfo player2 = (PlayerInfo)o2;
		return player1.position.compare(player1.position, player2.position);
	}

	public double overallDifference(PlayerInfo other)
	{
		return (position.differenceHeading(other.position) +
				position.differenceDistance(other.position));
	}
}
```

```
Position.java
package Common;

import java.io.PrintWriter;
import java.io.Serializable;
import java.util.Comparator;
import  EDU.gatech.cc.is.util.Vec2;

public class Position extends Vec2 implements Serializable, Comparator
{



   private final double TWO_PI = Math.PI * 2;

   public Position()
   {
         x =0.0;
      y = 0.0;

   }

   public Position(Position copyPosition)
   {
      x = copyPosition.x;
      y = copyPosition.y;
      r = copyPosition.r;
      t = copyPosition.t;

   }

   public Position(Vec2 copyPos)
   {
         x=copyPos.x;
         y=copyPos.y;
         r = copyPos.r;
         t = copyPos.t;

   }

   public Position(double initX, double initY)
   {
         x = initX;
         y=initY;

   }

   public boolean equals(Position newPosition)
   {
      boolean returnValue = false;
```

```
    if (newPosition == null)
    {
        return false;
    }
    if (x == newPosition.x && y == newPosition.y &&
        r == newPosition.r && t == newPosition.t)
    {
       returnValue = true;
    }
    return returnValue;
}

public boolean withinXMargin(Position compare)
{
        boolean result = false;
        if (Math.abs(x-compare.x) <= 0.2)
                result = true;

        return result;
}

public double distance(Position otherPosition)
{
        // pythagorean theorem
        double result =0;
        result = Math.abs(x-otherPosition.x) * Math.abs(x-otherPosition.x);
        result+= Math.abs(y-otherPosition.y) * Math.abs(y-otherPosition.y);
        result = Math.sqrt(result);

        return result;
}

public boolean withinMargin(Position compare)
{
        boolean result = false;
        if (distance(compare) <= 0.2)
                result = true;

        return result;
}

public double differenceXY(Position otherPosition)
{
        double difference = 0;

        // get absolute value of difference in row
        difference = distance(otherPosition);


        return difference;
}
```

```
public double differenceHeading(Position otherPosition)
{
        double difference = Math.abs(t- otherPosition.t);

        return difference;
}

public double differenceDistance(Position otherPosition)
{
        double difference = Math.abs(r-otherPosition.r);


        return difference;
}

public double overallDifference(Position otherPosition)
{
        double result = differenceHeading(otherPosition);
        result+=distance(otherPosition);
        return result;
}

        public int compare(Object o1, Object o2)
        {
                Position one = (Position)o1;
                Position two = (Position)o2;
                if (one.equals(two))
                {
                        return 0;
                }
                if (one.x > two.x)
                {
                        return 1;
                }
                else if (one.x < two.x)
                {
                        return -1;
                }
                else if (one.y > two.y)
                {
                        return 1;
                }
                else if (one.y < two.y)
                {
                        return -1;
                }
                else if (one.r > two.r)
                {
                        return 1;
                }
```

```
		else if (one.r < two.r)
		{
			return -1;

		}
		else if (one.t > two.t)
		{
			return -1;
		}
		else
			return 1;
	}

	public void toLog()
	{
		System.out.println("Position:\t" + x + "," +y);
		System.out.println("Heading:\t" + t);

	}
}
```

APPENDIX F – TEAMBOTS RUN-TIME AGENT CODE

CCxBRPlayer.java

```
/**
 * @(#)CCxBRPlayer.java
 *
 *
 * @author
 * @version 1.00 2010/11/13
 */
import  EDU.gatech.cc.is.util.Vec2;
import  EDU.gatech.cc.is.abstractrobot.*;
import  Common.*;


public class CCxBRPlayer extends ControlSystemSS
{
        private TeamContext tContext;
        private long        curr_time;              //What time is it?

        private Vec2        ball;


        private Vec2        center;                 // the center of the field
        private Vec2[]      teammates;              //Where are my teammates?
        private Vec2[]      opponents;              //Where are my opponents?
        private int         side;
        public String   displayString;

   public Situation current;
        /**
        Configure the control system.This method is
        called once at initialization time. You can use it
        to do whatever you like.
        */
        public void Configure()
        {
                // create team context
                tContext = TeamContext.getInstance();
                // get side - should be east =1
                curr_time = abstract_robot.getTime();
                if( abstract_robot.getOurGoal(curr_time).x < 0)
                        side = -1;
                else
                        side = 1;
        }


        /**
        Called every timestep to allow the control system to
        run.
        */
        public int TakeStep()
	{
		// Build current situation
		current = new Situation();
		// get the current time for timestamps

		curr_time = abstract_robot.getTime();

		boolean ballOnOurSide = false;

		/*--- Get some sensor data ---*/

		// get vector to the ball and to the center of the field
		ball = abstract_robot.getBall(curr_time);
		center = abstract_robot.getPosition(curr_time);
		current.myPosition.x = center.x;
		current.myPosition.y = center.y;
		current.myPosition.t = abstract_robot.getSteerHeading(curr_time);
//System.out.println("My position = " + center.x + "," + center.y);


		// transform to actual ball position
		// put ball into situation
		Vec2 actualBall = new Vec2(ball);
		actualBall.add(center);
		current.ballPosition.x = actualBall.x ;
		current.ballPosition.y = actualBall.y;
		current.ballPosition.t = ball.t;
		// calculate ball angle

		double temp = (double)(current.ballPosition.x - current.myPosition.x);
		double temp2 = (double)(current.ballPosition.y - current.myPosition.y);
		current.ballAngle = Math.atan2(temp2, temp );


		if (current.ballPosition.x < current.myPosition.x)
		{
			current.isBehindBall = true;
		}
		else
		{
			current.isBehindBall = false;
		}

		// get a list of the positions of our teammates
		teammates = abstract_robot.getTeammates(curr_time);
	//	System.out.println("Teammatest length = " + teammates.length);
		// place teammates into situation
		for (int i=0; i<teammates.length;i++)
		{
			Vec2 teamPosition = new Vec2(teammates[i]);
```

```
				current.teammates[i].position = new Position(teamPosition);
			}

			/* get a list of the positions of the opponents */
			opponents = abstract_robot.getOpponents(curr_time);
//			System.out.println("Opponents length = " + opponents.length);
			for (int i=0; i<opponents.length;i++)
			{
				Vec2 teamPosition = new Vec2(opponents[i]);


				current.opponents[i].position = new Position(teamPosition);
			}
			current.sortArrays();

			current.myNum = abstract_robot.getPlayerNumber(curr_time);
			current.side = side;

			Constants.States myContext = Constants.States.MOVING_0;
			// determine context

			if (current.isBehindBall  )
			{
				switch (current.myNum)
				{
				case 0:
					myContext = Constants.States.MOVING_0;
					break;
				case 1:
					myContext = Constants.States.MOVING_1;
					break;
				case 2:
					myContext = Constants.States.MOVING_2;
					break;
				case 3:
					myContext = Constants.States.MOVING_3;
					break;
				case 4:
					myContext = Constants.States.MOVING_4;
					break;
				}
			}
			else if (!current.isBehindBall )
			{
				switch (current.myNum)
				{
				case 0:
					myContext = Constants.States.GET_BEHIND_1;
					break;
				case 1:
					myContext = Constants.States.GET_BEHIND_1;
```

```
                        break;
                case 2:
                        myContext = Constants.States.GET_BEHIND_2;
                        break;
                case 3:
                        myContext = Constants.States.GET_BEHIND_3;
                        break;
                case 4:
                        myContext = Constants.States.GET_BEHIND_4;
                        break;
                }
        }
        current.myState = myContext;
        // call context and perform appropriate action
        Context context = tContext.translateNameToContext(myContext);
        //abstract_robot.setDisplayString(current.myNum + " " +myContext.toString());
        displayString = current.myNum + " " +myContext.toString();

        current.sortArrays();


        context.step(this, current);

        return 1;
    }



}
```

TeamContext.java
```
import Common.Constants;
import Common.Situation;
import Common.Position;

// team context for CCXBR framework
// author: Cynthia Johnson
public class TeamContext implements Constants
{
      private static boolean initialized = false;
      private static TeamContext instance = null;


      // initialize all the contexts in this game
      private              Context              closest=              new
Context(stateStrings[States.CLOSEST.ordinal()]);
      private            Context            moving0           =           new
Context(stateStrings[States.MOVING_0.ordinal()]);
      private            Context            moving1           =           new
Context(stateStrings[States.MOVING_1.ordinal()]);
      private            Context            moving2           =           new
Context(stateStrings[States.MOVING_2.ordinal()]);
      private            Context            moving3           =           new
Context(stateStrings[States.MOVING_3.ordinal()]);
      private            Context            moving4           =           new
Context(stateStrings[States.MOVING_4.ordinal()]);
      private           Context          getBehind0          =          new
Context(stateStrings[States.GET_BEHIND_0.ordinal()]);
      private           Context          getBehind1          =          new
Context(stateStrings[States.GET_BEHIND_1.ordinal()]);
      private           Context          getBehind2          =          new
Context(stateStrings[States.GET_BEHIND_2.ordinal()]);
      private           Context          getBehind3          =          new
Context(stateStrings[States.GET_BEHIND_3.ordinal()]);
      private           Context          getBehind4          =          new
Context(stateStrings[States.GET_BEHIND_4.ordinal()]);


      private TeamStates teamContext = TeamStates.DEFENSE;
      States blues[] = new States[4];

      private TeamContext()
      {


      }

      public static TeamContext getInstance()
      {
            // method to allow agents to get instance of team context
            if (!initialized)
            {
                  instance = new TeamContext();
                  initialized = true;
            }

            return instance;
}

public void destroy()
{
	instance = null;
	initialized = false;
}



Context translateNameToContext(States contextName)
{

	switch (contextName)
	{
	case CLOSEST:
		return closest;

	case GET_BEHIND_0:
		return getBehind0;

	case GET_BEHIND_1:
		return getBehind1;

	case GET_BEHIND_2:
		return getBehind2;

	case GET_BEHIND_3:
		return getBehind3;

	case GET_BEHIND_4:
		return getBehind4;

	case MOVING_0:
		return moving0;

	case MOVING_1:
		return moving1;

	case MOVING_2:
		return moving2;

	case MOVING_3:
		return moving3;

	case MOVING_4:
		return moving4;




	}
	return moving1;
}
```

```
}
```

Context.java
```
import java.io.EOFException;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.util.Collection;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Set;
import java.util.Random;
import java.util.Vector;

import Common.*;
import EDU.gatech.cc.is.util.Units;
import  EDU.gatech.cc.is.util.Vec2;
import  EDU.gatech.cc.is.abstractrobot.*;

public  class Context implements Common.Constants
{
	protected String name;
	protected HashMap<Situation, Constants.ACTION_PRIMITIVE>   policy;


	public Context(String contextName)
	{
		name= contextName;
		initializePolicy(name+".map");
	}


	public void step(CCxBRPlayer owningAgent, Situation currentSituation)
	{

		// check Hashmap for current situation

			// find nearest neighbor

			/*
			#get Collection of keys contained in HashMap using
			#  Collection values() method of HashMap class
			# */

		/*	if (this.name.equals(stateStrings[States.SEARCH_BOT_LEFT.ordinal()]))
			{
			System.out.println("In go to left");
			}*/
			 Set set= policy.keySet () ;

			//obtain an Iterator for Collection

			Iterator itr = set.iterator();
```

```
                        //iterate through HashMap values iterator
                        Constants.ACTION_PRIMITIVE actions = null;
          double lowScore = 1000;
          Situation lowScorer = null;

          int lowScorerCount = 0;
          int count = 0;

           Vector potentialActions = new Vector();
           int matchCount = 0;
           int misMatchCount = 0;
                        while(itr.hasNext())
                        {
                             Situation mySituation = (Situation)itr.next();


                             double                                    score                                    =
currentSituation.contextuallyWeightedNearestNeighbor(mySituation, currentSituation.myState);

                          if (score == lowScore)
                          {

                             Constants.ACTION_PRIMITIVE                    actionCheck                    =
(Constants.ACTION_PRIMITIVE)policy.get(mySituation);
                             if (!(actionCheck ==actions))
                             {
                                  boolean inVector = false;
                               for (int l=0; l< potentialActions.size();l++)
                               {
                                   ActionCount entry = (ActionCount)potentialActions.get(l);
                                   if ((actionCheck == entry.actions))
                                   {
                                        potentialActions.remove(entry);
                                        entry.count ++;
                                        potentialActions.add(entry);
                                        inVector = true;
                                        break;
                                   }
                               }
                               if (!inVector)
                               {
                                  ActionCount entry = new ActionCount();
                                  entry.actions = actionCheck;
                                  entry.count = 1;
                                  potentialActions.add(entry);
                               }
                                  misMatchCount ++;

                             }
                             else
                        {
                                matchCount ++;
                        }

                  }
                        if (score < lowScore)
                        {
                                lowScore = score;

                                lowScorer= mySituation;
                                actions                                                         =
(Constants.ACTION_PRIMITIVE)policy.get(lowScorer);
                                lowScorerCount = count;
                                matchCount = 0;
                                misMatchCount =0;
                                potentialActions = new Vector();

                        }
                        count ++;
                }
                int compareCount =matchCount;
                Constants.ACTION_PRIMITIVE chosenActions  = actions;
                if (misMatchCount > 0 && matchCount < misMatchCount)
                {

                        // find most prominant action
                        for (int l=0; l<potentialActions.size();l++)
                        {
                                ActionCount entry = (ActionCount)potentialActions.get(l);
                                if (entry.count >compareCount)
                                {
                                        chosenActions = entry.actions;
                                        compareCount = entry.count;
                        //                      System.out.println("Replaced actions");
                                }
                        }
                }

                // set anything necessary in calling Agent
                boolean performed = false;
                boolean badAction = false;
                if (chosenActions != null)
                {


                        Constants.ACTION_PRIMITIVE action = chosenActions;

                        performed = performAction(action,  owningAgent);
                        if (!performed)
                        {
                                badAction = true;
```

```
                    //      System.out.println("Bad      Action      for      "      +
owningAgent.myType.toString());

                    }


               }

          }



     private  boolean  performAction(Constants.ACTION_PRIMITIVE  action,    CCxBRPlayer
owningAgent)
     {
          boolean actionPerformed = true;
          SocSmallSim abstractRobot =owningAgent.getAbstractRobot();
          long currTime = abstractRobot.getTime();
          // Apply action to abstractRobot
          abstractRobot.setDisplayString(owningAgent.displayString + " "+ action.toString());

          if (action == ACTION_PRIMITIVE.GET_BEHIND_BALL)
          {
               // move to spot slightly above and beyond ball
               Vec2 ball = abstractRobot.getBall(currTime);

    // get vector to halfway between ball and goal

               Vec2 ourGoal = abstractRobot.getOurGoal(currTime);
               // see if ball between us and goal
               double tempdir = -Units.BestTurnRad(ourGoal.t,
                         ball.t);
               if (Math.abs(tempdir) < (Math.PI/2) )
               {
                    // yes
                    // swirl around ball
                    if (tempdir < 0)
                         ball.sett(ball.t - Math.PI/2);
                    else
                         ball.sett(ball.t + Math.PI/2);
               }
               abstractRobot.setSteerHeading(currTime, ball.t);
               abstractRobot.setSpeed(currTime, 1.0);

          }
          else if (action == ACTION_PRIMITIVE.MOVE_TO_BACKFIELD)
          {
               Vec2 ourGoal = abstractRobot.getOurGoal(currTime);
               // are we within 0.2 of goal
               if (ourGoal.r < 0.2)
               {
                    // yes = stop
                    abstractRobot.setSpeed(currTime,0);

                }
                else if (ourGoal.r > 0.3)
                {
                    // move toward goal full speed if more than
                    // 0.3 away
                    abstractRobot.setSteerHeading(currTime,ourGoal.t);
                    abstractRobot.setSpeed(currTime, 1.0);

                }
                else // somewhere inbetween
                {
                    abstractRobot.setSteerHeading(currTime,ourGoal.t);
                    // choose slow speed toward goal
                    double speed =  (ourGoal.r - 0.2)/0.1;
                    abstractRobot.setSpeed(currTime, speed);
                }
            }
            else if (action == ACTION_PRIMITIVE.MOVE_TO_BALL)
            {
                Vec2 ball = abstractRobot.getBall(currTime);
                if (!abstractRobot.canKick(currTime))
                {
                    // not close enough to kick
                    abstractRobot.setSteerHeading(currTime, ball.t);
                    abstractRobot.setSpeed(currTime, 1.0);
                }
                else
                {
                    //close enough to kick
                    // point towrd opposite goal and kick
                    Vec2 goal = abstractRobot.getOpponentsGoal(currTime);
                    abstractRobot.setSteerHeading(currTime, goal.t);
                    abstractRobot.kick(currTime);
                    abstractRobot.setSpeed(currTime, 0.5);

                }

            }

  return actionPerformed;

}



private void initializePolicy(String fileName)
{
            ObjectInputStream input = null;
```

```
          try // open file
      {
        input = new ObjectInputStream(
          new FileInputStream( fileName ) );
      } // end try
      catch ( IOException ioException )
      {
        System.err.println( "Error opening file." );
      } // end catch

            // read in single HashMap of policy generated by
      // learning algorithm
      try // input the values from the file
      {
            Object tempPolicy = input.readObject();
            if (tempPolicy instanceof HashMap<?,?>)
            {
                  policy = (HashMap<Situation, Constants.ACTION_PRIMITIVE>)tempPolicy;
            }


      } // end try
      catch ( EOFException endOfFileException )
      {
       System.out.println("end of file reached- nothing read");
      } // end catch
      catch ( ClassNotFoundException classNotFoundException )
      {
            classNotFoundException.printStackTrace();
        System.err.println( "Unable to create object." );
      } // end catch
      catch ( IOException ioException )
      {
            ioException.printStackTrace();
        System.err.println( "Error during reading from file." );
      } // end catch

  }

  //method to check if actions are equal
  private boolean actionsSame(Action compareTo, Action compareAgainst)
  {
          boolean same = false;
          if (compareTo!= null && compareAgainst != null)
          {
                  if (compareTo.equals(compareAgainst))
                  {
                          same = true;

                  }
          }
		return same;

	}


	// count of actions
	private class ActionCount
	{
		public Constants.ACTION_PRIMITIVE actions = null;
		public int count = 0;
	}

}
```

APPENDIX G: TEAMBOTS TRAINING LOG EXAMPLE

```
Simtime:            150
West Player Num:          1
force: 943, -179
Position:           -0.47179944866997425 0.006726016850005653
Heading:            0.6283185307179586
Speed:  0.5
West Player Num:          2
force: 650, -58
Position:           -0.1296391780932033   0.4786319522074346
Heading:            5.254788967838833
Speed:  0.5
West Player Num:          3
0 {} -> 0
Position:           -0.10500000000000001 8.279243814703243E-18
Heading:            2.5182511086550745E-16
Speed:  1.0
West Player Num:          4
force: 650, 59
Position:           -0.129679030353685    -0.47860810128612646
Heading:            1.0345889169854128
Speed:  0.5
West Player Num:          0
force: 9113, -178
Position:           -1.171799448669974    0.006726016850005653
Heading:            0.6283185307179586
Speed:  0.5
East Player Num:          1
move to ball
Position:           0.46971585360860924  0.24272644902562826
Heading:            3.1012852357677674
Speed:  0.5565695831417508
East Player Num:          2
move to ball
Position:           0.12108029045933653  0.47475973540040445
Heading:            3.918891207450441
Speed:  0.7770624821186477
East Player Num:          3
move to ball
Position:           0.46983390007558673  -0.24277365130138612
Heading:            3.1869577937501816
Speed:  0.5526480897515875
East Player Num:          4
move to ball
Position:           0.1210805380236585    -0.474759128533849
Heading:            2.364242091994057
Speed:  0.7770790940491115
East Player Num:          0
move to backfield
Position:           1.1557342698229496    0.00457586630152627
Heading:            3.145551904873635
Speed:  1.0
```

```
Ball Pos:       0.0     0.0
Ball Heading:   0.0
Score:  0       0
After:
Player Number:  1
move to ball
Heading:        3.1012852357677674
Speed:  0.5440002345838894
Kicked: false
Player Number:  2
move to ball
Heading:        3.918891207450441
Speed:  0.767530279364854
Kicked: false
Player Number:  3
move to ball
Heading:        3.1869577937501816
Speed:  0.5399778840878489
Kicked: false
Player Number:  4
move to ball
Heading:        2.364242091994057
Speed:  0.767587803740367
Kicked: false
Player Number:  0
move to backfield
Heading:        3.145551904873635
Speed:  0.8282914053303387
Kicked: false
```

REFERENCES

AgentBuilders. (2006, May 22). Retrieved February 19, 2010, from www.agentbuilder.com

Ahmad, H. F., Suguri, H., Ali, A., Malik, S., Mugal, M., Shafiq, M. O., et al. (2005). Scalable fault tolerant Agent Grooming Environment - SAGE. *Proceedings of the Fourth International Joint Conference on Autonomous Agents and Multi agent Systems (AAMAS).* Utrecht, Netherlands.

Althoff, K.-D. (2001). Case-Based Reasoning. In S. K. Chang, *Handbook on Software Engineering and Knowledge Engineering, Volume 1.* World Scientific.

Anderson, J. R. (1993). *The rules of the mind.* Hillsdale, NJ: Lawrence Erlbaum.

Argall, B. D., Browning, B., & Veloso, M. (2009). Learning robot motion control from demonstration and human advice. *AAAI's Spring Symposia* (pp. 7-15). AAAI.

Argall, B. D., Chernova, S., Veloso, M., & Browning, B. (2009). A survey of robot learning from demonstration. *Robotics and Autonomous Systems , 57*, 469-483.

Atkeson, C. G., Moore, A., & Schaal, S. (1997). Locally weighted learning for control. *Artificial Intelligence Review , 11*, 75-113.

Balch, T., & Various. (2000). *TeamBots*. Retrieved March 15, 2009, from TeamBots: www.teambots.org

Baron, S., Zacharias, G., Muralidharan, R., & Lancraft, R. (1980). PROCRU: A model for analyzing flight crew procedures in approach to landing. *Proceedings of the Eight IFAC World Congress.* Tokyo, Japan.

Barrett, G. C. (2007). Collaborative context-based reasoning. *Doctoral Dissertation* . Orlando, FL: University of Central Florida.

Basilico, J., Benz, Z., & Dixon, K. R. (2008). The Cognitive Founder: A flexible platorm for intelligent agent modeling. *Proceedings of BRIMS 2008.*

Baum, L. E., Petrie, T., Soules, G., & Weiss, N. (1970). A maximization technique occurring in the statistical analysis of probabilistic functions of markov chains. *Annals Mathematical Statistics , 41* (1), 164-171.

Bazire, M., & Brezillon, P. (2005). Understanding context before using it. *Proceedings of the 5thInternational and Interdisciplinary Conference on Context* (pp. 29-40). Paris, France: Springer Berlin.

Bellifemine, F. L., Caire, G., & Greenwood, D. (2007). *Developing Multi-Agent Systems with JADE* . West Sussex, England: Wiley.

Bellifemine, F., Poggi, A., & Rimassa, G. (2001). Developing multi-agent systems with a FIPA-compliant agent framework. *SOFTWARE—PRACTICE AND EXPERIENCE , 31*, 103-128.

Bentivegna, D. C. (2004). *Learning from observation using primitives - a doctoral dissertation.* Atlanta, GA: Georgia Institute of Technology.

Bentivegna, D. C., & Atkeson, C. G. (2001). Learning from observation using primitives. *IEEE International Conference on Robotics and Automation,,* (pp. 1988-1993). Seoul, Korea.

Best, B. J., Dixon, K. R., Speed, A., & Fleetwood, M. D. (2008). Modeling an Unstructured Driving Domain: A Comparison of Two Cognitive Frameworks. *Proceedings of the 17th Conference on Behavior Representation in Modeling Simulation.*

Bratman, M. E. (1992). Shared cooperative activity. *The Philisophical Review , 101* (2), 327-341.

Bratman, M. E. (1993). Shared intention. *Ethics , 104* (1), 97-113.

Brezillon, P. (2003). Context-based Modeling of Operators' Practices by Contextual Graphs. *Human Centered Processes : 14th Mini Euro Conference, Proceedings of.* Luxembourg.

Brezillon, P. (2003). Representation of procedures and practices in contextual graphs. *The Knowledge Engineering Review , 18* (2), 147-174.

Byrne, R. W., & Russon, A. E. (1998). Learning by imitation: A hierarchical approach. *Behavioral and Brain Sciences , 21*, 667-684.

Cannon-Bowers, J. A., Salas, E. C., & Converse, S. A. (2001). Shared mental models in expert team decision making. In R. J. Sternberg, & E. L. Grigorenko, *Environmental effects on cognitive abilities* (pp. 221-246). Mahwah, NJ: L. Erlbaum Associates.

Chalupsky, H., Gil, Y., Knoblock, C. A., Lerman, K., Oh, J., Pynadath, D. V., et al. (2002). Electric Elves: Agent Technology for supporting human organizations. *AI Magazine , 23* (2), 11-24.

Chernova, S., & Veloso, M. (2007). Confidence-Based Policy Learning from Demonstration Using Gaussian Mixture Models. *Proceedings of AAMAS'07, the Sixth International Joint Conference on Autonomous Agents and Multi-Agent Systems.* Honolulu, HI.

Chernova, S., & Veloso, M. (2008). Teaching Multi-Robot Coordination using Demonstration of Communication and State Sharing. *Proceedings of AAMAS'08, the Seventh International Joint Conference on Autonomous Agents and Multi-Agent Systems.* Estoril, Portugal.

Cohen, P. R., & Levesque, H. J. (1991). Teamwork. *Nous , 104* (1), 487-512.

da Silva, B. N. (2009). Learning from disagreeing demonstrators. *AAAI Spring Symposia* (pp. 36-39). AAAI.

Deutsch, S. E., Macmillan, J., Cramer, N. L., & Chopra, S. (1997). *Operator Model Architecture (OMAR) Final Report; BBN Report No 8179.* Cambridge, MA: BBN Corporation.

Dillmann, R., Kaiser, M., & Ude, A. (1995). Acquisiton of elementary robot skills from human demonstration. *International Symposium on Intelligent Robotic Systems.*

Drewes, P. (1997). Automated student performance monitoring in training simulation . *Doctoral Dissertation* . Orlando, FL: University of Central Florida.

Druskat, V. U., & Pescosolido, A. T. (2002). The content of effective teamwork mental models in self-managing teams: Ownership, learning and heedful interrelating. *Human Relations , 55* (3), 283-314.

Endsley, M. R. (1995). Toward a Theory of Situation Awareness in Dynamic Systems. *Human Factors , 37* (1), 32-64.

Evertsz, R., Ritter, F. E., Busetta, P., Pedrotti, M., & Bittner, J. L. (2008). CoJACK – Achieving principled behaviour variation in a moderated cognitive architecture. *Proceedings of BRIMS 2008.*

Fernlund, H. K. (2004). Evolving models from observed human performance. *Doctoral Dissertation* . Orlando, FL: University of Central Florida.

Fernlund, H., Gonzalez, A. J., Ekblad, J., & Rodriguez, A. (2009). Trainee Evaluation through after-action review by comparison. *The Journal of Defense Modeling and Simulation* .

Floyd, M. W., Esfandiari, B., & Lam, K. (2008). A Case-based Reasoning Approach to Imitating RoboCup Players. *Proceedings of the Twenty-First International FLAIRS Conference*, (pp. 251-256).

Forsythe, C., & Xavier, P. G. (2006). Cognitive models to cognitive systems. In C. Forsythe, M. L. Bernard, & T. E. Goldsmith, *Cognitive Systems: Human Cognitive Models in Systems Design.* Mahwah, NJ: Lawrence Erlbaum Associates.

Galef, J. B., & Giraldeau, L. (2001). Social influences on foraging in vertbrates: Causal mechanisms and adaptive functions. *Animal Behaviour , 61*, 3-15.

Georgeff, M., Pell, B., Pollack, M., Tambe, M., & Woolridge, M. (1999). The belief-desire-intention model of agency. *Proceedings of Agents, Theories, Architectures and Languages (ATAL).*

Gerber, W. J. (2001). Real-time synchronization of behavioral models with human performance in a simulation . *Doctoral Dissertation* . Orlando, FL: University of Central Florida.

Giunchiglia, F. (1993). Contextual Reasoning. *Epistemologia - Special Issue on I Linguaggi e , 16*, 345-364.

Glenn, F. A., Schwartz, S. M., & Ross, L. V. (1992). *Development of a Human Operator Simulator Version V (HOS-V): Design and implementation.* Alexandria, VA: U.S. Army Research Institute for the Behavioral and Social Sciences.

Gonzalez, A. J., Gerber, W. J., & Castro, J. (2002). Automated acquisition of tactical knowledge through contextualization. *Computer Generated Forces and Behavior Representation Conference.*

Gonzalez, A. J., Stensrud, B. S., & Barrett, G. (2008). Formalizing Context-Based Reasoning - A Modeling Paradigm for Representing Tactical Human Behavior. *International Journal of Intelligent Systems , 23* (7), 822-847.

Grosz, B. J., & Hunsberger, L. (2006). The Dynamics of Intention in Collaborative Activity. *Cognitive Systems Research , 7* (2-3), 259-272.

Grosz, B. J., & Kraus, S. (1996). Collaborative plans for complex group action. *Artificial Intelligence , 86* (2), 269-357.

Grosz, B. J., & Kraus, S. (1999). The Evolution of SharedPlans. In M. Wooldridge, & A. Rao, *Foundations of Rational Agency* (pp. 227-262). Kluwer.

Hadad, M., & Kraus, S. (1999). SharedPlans in Electronic Commerce. In M. Klusch, *Intelligent Information Agents* (pp. 204-230). Berlin: Springer-Verlag.

Han, K., & Veloso, M. (2000). Automated robot behavior recognition. In J. Hollerbach, & D. Koditschek, *Robotics Research: the Ninth International Symposium* (pp. 199--204). London: Springer-Verlag.

Henninger, A. E. (2001). *Neural Network Based Movement Models to Improve the Predictive Utility of Entity State Synchronization Methods for Distributed Simulations.* Orlando, FL: University of Central Florida Doctoral Dissertation.

Hespanha, J. P., Prandini, M., & Sastry, S. (2000). Probabilistic Pursuit-Evasion Games: A One-Step Nash Approach. *Proceedings of the 39th IEEE Conference on Decision and Control* (pp. 2272-2277). Sydney, Australia: IEEE.

Hill, R., Chen, J., Gratch, J., Rosenbloom, P., & Tambe, M. (1997). Intelligent agents for the synthetic battlefield: A company of rotary wing aircraft.

Howden, N., Ronnquist, R., Hodgson, A., & Lucas, A. (2001). JACK Intelligent Agents - Summary of an Agent Infrastructure. *Proceedings of the 5th ACM International Conerence on Autonomous Agents.*

Hsu, C.-L., Chou, S.-F., Tsay, J.-C., & Wang, F.-J. (2003). JTEAM: A Framework for Effective Teamwork of BDI-based Agents. *Proceedsing of The Ninth IEEE Workshop on Future Trends of Distributed Computing Systems.*

Intille, S. S., & Bobick, A. F. (1999). A Framework for Recognizing Multi-Agent Action from Visual Evidence. *Proceedings of the National Conference on Artificial Intelligence.* AAAI.

Isaac, A., & Sammut, C. (2003). Goal-directed learning to fly. *Proceedings of the International Conference on Machine Learning*, (pp. 258-265). Washington, D.C.

JADE- Java Agent DEvelopment framework. (n.d.). Retrieved November 4, 2009, from JADE- Java Agent DEvelopment framework: http://jade.tilab.com/

Jennings, N. R. (1995). Controlling cooperative problem solving in industrial multi-agent systems using joint intentions. *Artificial Intelligence , 75* (2), 195-240.

Jiang, C., & Sheng, C. (2009). Case-based reinforcement learning for dynamic inventory control in a multi-agent supply-chain system. *Expert Systems with Applications , 36*, 6520–6526.

Johansson, L. (1999). Cooperating AIPs in the context-based reasoning paradigm. *Master's Thesis, ECE Dept.* Orlando, FL: University of Central Florida.

Johnson-Laird, P. N. (1983). *Towards a cognitive science of language, inference and consciousness.* Cambridge, MA: Harvard University Press.

Jones, R. M., Laird, J. E., & Nielsen, P. E. (1996). Moving intelligent automated forces into theater-level scenarios. . *Proceedings of the Sixth Conference on Computer Generated Forces and Behavioral Representation*, (pp. 113-117). Orlando, FL.

Juang, B. H., & Rabiner, L. R. (1990). The segmental k-means algorithm for estimating parameters of hidden Markov models. *IEEE Transactions on Acoustics, Speech and Signal Processing , 38* (9), 1639–1641.

Kalyanakrishnan, S., Liu, Y., & Stone, p. (2009). Half Field Offense in RoboCup Soccer: A Multiagent Reinforcement Learning Case Study . In J. G. Carbonell, & J. Seikman, *RoboCup 2006: Robot Soccer World Cup X.* Berlin/Heidleberg: Springer.

Kennedy, J., & Eberhart, R. (1995). Particle Swarm Optimization. *Proceedings of the 1995 IEEE Conference on Neural Networks.*

Kitano, H., Asada, M., Kuniyoshi, Y., Noda, I., Osawa, E., & Matsubara, H. (1997). RoboCup: A challenge problem for AI. *AI Magazine , 18* (1), 73-85.

Kokinov, B. (1999). Dynamics and Automaticity of Context:A Cognitive Modeling Approach. *Proceedings of Second International and Interdisciplinary Conference,CONTEXT'99*, (pp. 200-213). Trento. Italy.

Kokinov, B., & Yoveva, M. (1996). Context effects on problem solving. *Procedeedings of the Eighteenth Annual Conference of the Cognitive Science Society.* Hillsdale, NJ: Erlbaum.

Laird, J. E. (2008). Extending the SOAR cognitive architecture. In P. Wang, & S. Franklin, *Proceedings of the first Artificial General IntelligenceConference* (pp. 224-234). IOS Press.

Laird, J. E., Newell, A., & Rosenbloom, P. S. (1987). SOAR: An architecture for general intelligence. *Artificial Intelligence , 33* (1), 1-64.

Laughery, K. R., & Corker, K. M. (1997). Computer modeling and simulation of human/system performance. In G. Salvendy, *Handbook of Human Factors.* New York, NY: John Wiley and Sons.

Lee, H., Mihailescu, P., & Shepherdson, J. (2007). Realizing Teamwork in the Field: An Agent-Based Approach. *Pervasive Computing* , 85-92.

Leng, J., Li, J., & Jain, L. C. (2008). A Role-Based Framework for Multi-agent Teaming. *Knowledge-Based Intelligent Information and Engineering Systems: 12th International Conference Proceedings, Part III* . Zagreb, Croatia: Springer Berlin.

Loxley, A. (1997). *Collaboration in health and welfare.* London: Jessica Kingsley Publishers.

Luotsinen, L. J., Fernlund, H., & Boloni, L. (2007). Automatic Annotation of Team Actions in Observations of Embodied Agents. *Proceedings of The Sixth Intl. Joint Conf. on Autonomous Agents and Multi-Agent Systems (AAMAS 07),* (pp. 32-24).

Meyer, D. E., & Kieras, D. E. (1997). A computational theory of executive cognitive processes and multiple task performance. Part 1. Basic Mechanisms. *Psychological Review , 104*, 2-65.

Mickan, S., & Rodger, S. (2000). Characteristics of effective teams: A literature review. *Australian Health Review , 23* (2), 201-208.

Mohammed, S., Klimoski, R., & Rentsch, J. R. (2000). The measurement of team mental models: We have no shared schema. *Organizational Research Methods , 3*, 123-165.

Molineaux, M., Aha, D. W., & Sukthankar, G. R. (2009). Beating the Defense: Using Plan Recognition to Inform Learning Agents. *Proceedings of the Twenty-Second Annual Conference of the Florida Artificial Intelligence Research Society.* AAAI Press.

Ng, A. Y., Harada, D., & Russell, S. (1999). Policy invariance under reward transformations: theory and application to reward shaping. *ICML'99,* (pp. 278-287).

Nguyen, M. H., & Wobcke, W. (2006). A flexible framework for SharedPlans. In *AI 2006: Advances in Artificial Intelligence* (pp. 393-402). Berlin: Springer-Verlag.

Nitschke, G. (2003). Co-evolution of cooperation in a pursuit evasion game . *Proceedings of 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems* (pp. 2037 - 2042). IEEE.

Nwana, H. S., Ndumu, D. T., Lee, L. C., & Callis, J. C. (1999). Zeus: A toolkit for Building Distributed Multiagent Systems. *Applied Artificial Intelligence , 13*, 129-185.

Ontañón, S., Bonnette, K., Mahindrakar, P., Gomez-Martin, M. A., Long, K., Radhakrishnan, J., et al. (2009). Learning from Human Demonstrations for Real-Time Case-Based Planning. *IJCAI-09 Workshop on Learning Structural Knowledge From Observations.*

Ortiz, C. L., & Grosz, B. J. (2002). Interpreting Information Requests in Context: A Collaborative Interface for Distance Learning. *Autonomous Agents and Multi-Agent Systems , 5*, 429-465.

Pew, R., & Mavor, A. (1998). *Modeling Human and Organization Behavior: Application to Military Simulations.* Washington, D.C. : National Academy Press.

Pomerlau, D. (1991). Efficient training of artificial neural networks for autonomous navigation. *Neural Computation , 3* (1), 88-97.

Poslad, S., Buckle, P., & Hadingham, R. (2000). The FIPA-OS agent platform: Open source for open standards. *Proc of the Fifth International Conference and Exhibition on the Practial Application of Intelligent Agents and Multi-Agents.* Manchest, UK.

Price, B., & Boutilier, C. (2003). Accelerating Reinforcement Learning through Implicit Imitation. *Journal of Artificial Intelligence Research , 19*, 569-629.

Proenza, R. (1997). *A framework for multiple agents and memory recall within the context-based reasoning paradigm.* Orlando: University of Central Florida.

Qureshi, M. A. (2001, July). The Evolution of Agents. *Doctoral Dissertation* . London, England: University of London.

Rich, C., & Sidner, C. L. (1998). COLLAGEN: A Collaboration Manager for Software Interface Agents. *User Modeling and User-Adapted Interaction , 6*, 315-350.

Riedmiller, M., Gabel, T., Hafner, R., & Lange, S. (2009). Reinforcement learning for robot soccer. *Autonomous Robots , 27*, 55-73.

Russell, S., & Norvig, P. (2003). *Artificial Intelligence: A Modern Approach.* Upper Saddle River, NF: Prentice Hall.

Ryder, J., & Zachary, W. (1991). Experimental validation of the attention switching component of the COGNET framework. *Proceedings of Human Factors Society 35th Annual Meeting* (pp. 72-76). Santa Monica, CA: Human Factors and Ergonomics Society.

Sadik, S., Ali, A., Ahmad, H. F., & Sugari, H. (2006). Using honey bee teamwork strategy in software agents. *10th International Conference on CSCW in Design*, (pp. 620-626). China.

Salas, E., Shawn Burke, C., & Cannon-Bowers, J. A. (2000). Teamwork: Emerging Principles. *International Journal of Management Reviews , 2* (4), 339-356.

Sammut, C., Hurst, S., Kedzier, D., & Michie, D. (1992). Learning to fly. *Proceedings of the 9th International Machine Learning Conference.* Aberdeen, Scotland: Morgan Kaurmann.

Saunders, J., Nehaniv, C. L., & Dautenhahn, K. (2006). Teaching robots by moulding behavior and scaffolding the environment. *Proceedings of the 1st ACM/IEEE International Conference on Human-Robot Interation.*

Scerri, P., Pynadath, D. V., Johnson, L., Rosenbloom, P., Schurr, N., Si, M., et al. (2003). A prototype infrastructure for distributed robot-agent-person teams. *The Second International Joint Conference on Autonomous Agents and MultiAgent Systems.*

Schurr, N., Maheswaren, R., Scerri, P., & Tambe, M. (2006). From STEAM to Machinetta: The evolution of a BDI teamwork model. In R. Sun, *Cognition and Multi-Agent Inteaction* (pp. 307-328). New York: Cambridge University Press.

Schurr, N., Marecki, J., Lewis, J. P., Tambe, M., & Scerri, P. (2005). The DEFACTO System: Coordinating Human-Agent Teams for the Future of Disaster Response. In R. H. Bordini, M. Dastani, J. Dix, & A. El Fallah Seghrouchni, *Multi-Agent Programming.* Springer.

Shaw, M. L., & Gaines, B. R. (1983). A computer aid to knowledge engineering. *Proceedings of the British Computer Society Conference on Expert Systems.*

Sidani, T. A. (1994). Learning situational knowledge through observation of expert performance in a simulation-based environment. *Doctoral Dissertation* . Orlando, Florida: University of Central Florida.

Singh, M. P., Huhns, M. N., & Stephens, L. M. (1993). Declarative Representations of Multiagent Systems. *IEEE Transactions on Knowledge and Data Engineering , 5* (5), 721-739.

Stanley, K., & Miikkulainen, R. (2002). Evolving Neural Networks through augmenting topologies. *Evolutionary Computation , 10* (2), 99-127.

Stein, G. M. (2009, Summer). FALCONET:Force-feedback Approach for Learning from Coaching and Observation using Natural and Experiential Training. *Doctoral Dissertation* . Orlando, FL: University of Central Florida.

Stensrud, B. S., & Gonzalez, A. J. (2008). Discovery of High-Level Behavior From Observation of Human Performance in a Strategic Game. *IEEE TRANSACTIONS ON SYSTEMS, MAN, AND CYBERNETICS—PART B: CYBERNETICS , 38* (3), 855-874.

Stone, P. (2007). Multiagent learning is not the answer. It is the question. *Artificial Intelligence* , 402-405.

Stone, P., & Sutton, R. S. (2001). Scaling Reinforcement Learning toward RoboCup Soccer. *The Eighteenth International Conference on Machine Learning*, (pp. 537-544). Williamstown, MA.

Sukthankar, G., & Sycara, K. (2006). Robust Recognition of Physical Team Behaviors using Spatio-Temporal Models. *Proceedings of Fifth International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS).* Hakodate, Hokkaido, Japan: ACM.

Sundstrom, E. (1999). Chalenges of supporting work team effectiveness. In E. Sundstrom, *Supporting Work Team Effectiveness* (pp. 1-25). San Francisco, CA: Jossey-Bass Publishers.

Swartout, W., Gratch, J., Hill, R. W., Hovy, E., Marsella, S., Rickel, J., et al. (2006). Towards Virtual Humans. *AI Magazine , 27* (2), 96-108.

Sycara, K. (1998). Multiagent systems. *AI Magazine* , 79-91.

Tambe, M. (1995). Towards flexible teamwork. *Proceedings of the First International Conference on Multi-Agent Systems* (pp. 368-375). San Francisco: AAAI Press.

Tambe, M., Shen, W., Mataric, M., Pynadath, D., Goldberg, D., Modi, P. J., et al. (1999). Using TEAMCORE to make agents team-ready. *Proceedings of the AAAI Spring Symposium on Agents in Cyberspace.* Menlo Park: The AAAI Press.

Tambe, M., Shen, W.-M., Mataric, M., Pynadath, D. V., Goldberg, D., Modi, P. J., et al. (1999). Teamwork in Cyberspace: Using TEAMCORE to make agents team-ready. *AAAI Spring Symposium on agents in cyberspace* (pp. 136-141). The AAAI Press.

Trinh, V. (2009). *Contextualizing observational data for modeling human performance.* Orlando, FL: University of Central Florida Doctoral Dissertation.

Turner, R. M. (1993). Context-sensitive reasoning for autonomous agents and cooperative distributed problem solving. *Proceedings of the IJCAI Workshop on using Knowledge in Context.*

Turner, R. M. (1998). Context-mediated behavior for AI applications. *Proceedings of the 11th international Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems IEA/AIE-98, 1*, pp. 538-545. Castell, Spain.

Ude, A., Atkeson, C. G., & Riley, M. (2004). Programming full-body movements for humanoid robots by observation. *Robotics and Autonomous Systems , 47*, 93-108.

Underwood, G. D., & Bright, J. E. (1996). Cognition with and without awareness. In G. D. Underwood, *Implicit Cognition* (pp. 1-40). Oxford: Oxford University Press.

Velagapudi, P., Prokopyev, O., Sycara, K., & Scerri, P. (2007). Maintaining Shared Belief in a Large Multiagent Team. *Proceedings of FUSION '07.*

Vidal, R., Shakernia, O., Kim, H. J., Shim, D. H., & Sastry, S. (2002). Probabilistic Pursuit–Evasion Games: Theory,Implementation, and Experimental Evaluation. *IEEE Transaction on Robotics and Automation , 18* (5), 662-669.

Wang, X. (1995). Learning by observation and practice: An incremental approach for planning operator acquisition. *Proceedings of the 12th International Conference on Machine Learning.* Tahoe City, CA: The International Machine Learning Society.

White, C., & Brogan, D. (2006). The Self Organization of Context for Learning in Multiagent Games. *Proceedings of the Second Artificial Intelligence and Interactive Digital Entertainment Conference* (pp. 92-97). Marina Del Ray, CA: AAAI Press.

Xu, R., & Wunsch II, D. (2005). Survey of Clustering Algorithms. *IEEE Transactions on Neural Networks , 16* (3), 645-678.

Yen, J., Fan, X., Sun, S., Hanratty, T., & Dumer, J. (2006). Agents with shared mental models for enhancing team decision makings. *Decision Support Systems , 41*, 634-653.

Yong, C. H., & Mikkulainen, R. (2001). *Cooperative coevolution of multi-agent systems.* Austin, TX: University of Texas at Austin.

Zachary, W., Le Mentec, C., & Ryder, J. (1996). Interface agents in complex systems. In C. A. Ntuen, & E. H. Park, *Human interaction with complex systems: conceptual principles and design* (pp. 35-52). Norwell, MA: Kluwer Academic Publishers.

Zhang, C., Abdallah, S., & Lesser, V. (2009). Integrating organizational control into multi-agent learning. *Proceedings of the 8th International Conference on Autonomous Agents and Multiagent Systems,* (pp. 757-764). Budapest, Hungary.

Zibetti, E., Quera, V., Tijus, C., & Beltran, F. S. (2001). Reasoning based on categorisation for interpreting and acting: a first approach. *Mind & Society , 2* (2), 87-104.