Using RoboCup to Teach Multiagent Systems and the Distributed Mindset

José M. Vidal
Computer Science and Engineering
University of South Carolina
Columbia, SC 29208
vidal@sc.edu

Paul Buhler
Computer Science
College of Charleston
66 George Street
Charleston, SC 29424
pbuhler@cs.cofc.edu

Abstract

We present our experiences using the RoboCup soccerserver simulator and Biter, our own agent platform, for the teaching of a graduate multiagent systems' class. The RoboCup simulator and Biter are both described. We argue that the combination of RoboCup and Biter forms an effective platform for the teaching of multiagent systems and the distributed mindset. Results from two semesters using these tools are presented. These results confirm our claims. Finally, we characterize this work within the framework provided by the STEELMAN Draft of the Computing Curricula 2001 initiative.

1 Introduction

The RoboCup [2] simulation league tournament has proven to be successful at bringing together researchers from various areas of computer science and engineering such as artificial intelligence, multiagent systems, distributed programming, software engineering, and real-time systems engineering. The competition matches teams of simulated soccer players in a ten minute match. While the environment is simplified, it nonetheless incorporates many of the complexities one would expect in a robotic application such as friction, wind, real-time performance, limited and noisy inputs, noisy effectors, etc. The contest has proven to be extremely popular. The last competition, held jointly with IJCAI '01, drew over 40 teams in the simulation league.

This paper presents our experiences using RoboCup and Biter—our agent framework—to teach two consecutive graduate-level classes in multiagent systems. We argue that the combination of RoboCup and Biter forms an effective platform for the teaching of multiagent systems and the distributed mindset. Asking students to build RoboCup teams exposes them to the non-intuitive emergent consequences of simple local actions. The exercise also exposes the students to the often frustrating complexities of debugging distributed applications and teaches them strategies for overcoming these difficulties.

Proceedings of the 33^{rd} ACM Technical Symposium on Computer Science Education, 2002.

2 The Distributed Mindset

In [7] Resnick shows how humans have a tendency to erroneously attribute centralized control to many situations. For example, when children were asked to describe how ants achieve their foraging behaviors most of them attributed it to some form of centralized control stemming from the queen. Many adults also erroneously attribute too much centralized control to phenomena such as the economy ("It is Greenspan's fault."), corporations ("It is the CEO's fault."), and the Internet ("The web is down.").

Resnick argues that these misconceptions should be addressed as early as possible in a child's education. Our experience indicates that distributed, as opposed to hierarchical, control is a concept that is initially difficult for most CSE students to grasp. We believe that a combination of instruction and hands-on experimentation is required to help students develop an understanding of distributed systems. These experiences allow students to develop the appropriate intuitions which will let them design and debug distributed applications which exhibit complex emergent behaviors. This need is even more pronounced in today's highly networked world where web-based and peer-to-peer applications are quickly becoming the norm [6].

3 A RoboCup Overview

The RoboCup tournament started in 1997 and has been held every year since. It consists of several robotic leagues as well as one simulated league. Our research focuses solely on the simulated league. For the simulated

league the organizers provide a soccerserver application which simulates the physics of the playing field. The field is two-dimensional; the players and ball are perfect circles. Each player must be controlled by a completely independent process, that is, players on the same team cannot share global variables or use any form of inter-process communications. The players have a view-cone of limited radius. The server adds some noise to all the players' sensory inputs and actions. The noise is meant to simulate the type of restrictions one would expect if the players were robots.

The soccerserver updates its world model every 100ms. Each player can send at most one action to the server every 100ms. All communications are over UDP and use the soccerserver's communication protocol. The actions available to the players are *dash*, *turn*, *kick*, and *say*. Every 150ms each player receives a message from the server telling it what it sees, i.e., what is in it's view-cone, as well as what it hears. This information is given in polar coordinates with the player as the origin, and the zero angle points to where the agent is facing. For example, the server might tell a player that is 'sees' the ball at a distance of 1.3 meters and an angle of -10 degrees from where it is currently looking. Since the information from the server is all relative to the player's current location the only way a player can determine its location is by triangulating it from some fixed objects whose location is known. These fixed objects are a series of flags placed around the field, boundary lines, and the goal posts.

The player's *dash* and *kick* actions can be given a force parameter to indicate the force that should be given. In the case of the dash command the server updates the player's position by taking into account its current momentum and force. For the kick command, the server takes into account the player's and the ball's momentum as well as the relative angle of the ball to the player—it is easier to kick a ball that is directly in front of the player. In both cases the server adds some noise that is proportional to the difficulty of the action. In other words, neither player nor ball consistently end up where expected, thus complicating the task of implementing reliable players. The *say* command is used to yell any arbitrary string which will be heard by some of the nearby players. The *turn* command turns the player in the appropriate direction.

4 On Using RoboCup

The RoboCup simulator has many qualities which make it an excellent platform for the teaching of the distributed mindset and multiagent systems' design.

1. It presents a complex distributed environment which requires the coordination of many autonomous agents in order to win the game. Since the players have little direct communications with each other, a distributed solution is necessary.
2. It raises many soft real-time issues. The agents cannot spend too much time thinking.
3. It is a noisy domain. The agents that operate in it must be able to compensate for errors in their input.
4. It is a well-known problem. There is no need to spend time explaining and understanding a new problem domain.
5. The competitive aspect is a great motivator. We have found that many students are highly motivated by the prospect of defeating their classmates in a game of simulated soccer.
6. The international RoboCup initiative has generated a wealth of research materials that are easily located and consumed by students.

While all these characteristics make RoboCup a great platform, there are several aspects which make it hard to use for instructional purposes.

1. There is a large amount of low-level work that needs to be done before starting to develop coordination strategies. Specifically:
   (a) Any good player will need to parse the sensor input and create its own world map which uses absolute coordinates. That is, the input the agents receive has the objects coordinates as relative polar coordinates from the player's current position. While realistic, these are hard to use in the definition of behaviors. Therefore, a sophisticated player will need to turn them into globally absolute coordinates.
   (b) The players need to implement several sophisticated geometric functions that answer some basic questions like: "Who should I be able to see now?".
   (c) The players also need to implement functions that determine the argument values for their commands. For example: "How hard should I kick this ball so that it will be at coordinates x, y next time?".
2. It is hard to keep synchronized with the soccerserver's update loop. Specifically, the players have to make sure they send one and only one action for each clock "tick". Since the soccerserver is running on a different machine, the player has to make sure it keeps synchronized and does not miss action opportunities, even when messages are lost.
3. Students new to agent design need some guidance in establishing a basic agent architecture. They lack experience using techniques for balancing goal-driven and reactive behaviors.

These drawbacks forced students to spend most of their time writing code to handle the UDP message parsing and the construction of a world model, as we detail in Section 6. Therefore, we designed a basic RoboCup client, called Biter, that implements all the features the students will need in order to quickly get started testing new behaviors and coordination protocols.

5 The Biter Platform

Biter provides its users with an absolute-coordinate world model, a set of low-level ball handling skills, a set of higher-level skill based behaviors, and our Generic Agent Architecture (GAA) [9] which forms the framework for agent development. Additionally, many functional utility methods are provided which allow users to focus more directly on planning activities. Biter is written in Java 2. A complete description of Biter, its source code, Javadoc API, and UML diagrams are available [1].

5.1 Biter's World Model

In the RoboCup domain it has become clear that agents need to build a world model [8]. The Biter world model contains slots for both static and dynamic objects. Static objects have a field placement that does not change during the course of a game. These include flags, lines, and the goals. In contrast, dynamic objects move about the field during the game. These include the players and the ball. Static objects are held within a HashMap data structure, while dynamic objects are stored in an ArrayList. Both HashMap and ArrayList are provided as part of the Java 2 collection classes.

5.2 The Generic Agent Architecture

Practitioner's new to agent-oriented software engineering [4] often stumble when building an agent that needs both reactive and long-term behaviors, usually settling for a completely reactive system and ignoring multi-step behaviors. For example, in RoboCup an agent can take an action at every clock tick. This action can simply be a reaction to the current state of the world, or it can be dictated by a long-term plan. Simple agent implementations choose an action at each time step by executing a long series of if-then statements where the conditional only checks the value of recent inputs. Unfortunately, such implementations make it very hard to add multi-step behaviors. The usual strategy is to add a "mode" to the agent which is then used in the conditional part of the if-then statements to determine which action to take. This strategy, while functional, is not very elegant (it is not an object-oriented solution) and does not scale well with the number of multi-step behaviors.

Biter implements a GAA which provides the structure needed to guide users in the development of a solid object-oriented agent architecture. The GAA is designed for agents that receive input from the environment at discrete intervals and take discrete actions. That is, we envision an agent that receives readings from its sensors and takes actions using its effectors. This is a common method for modeling autonomous agents [10, Chapter 1] and captures many agent applications.

The GAA provides a mechanism for scheduling activities each time the agent receives some form of input. An activity is defined as a set of actions to be performed over time. The action chosen at any particular time might depend on the state of the world and the agent's internal state. The two types of activities we have defined are conversations and behaviors. Conversations are series of messages exchanged between agents. Behaviors are actions taken over a series of time steps. The ActivityManager determines which activity should be called to handle any new input. A general overview of the system can be seen in Figure 1.

5.3 Behaviors

Biter also provides users with the most basic, and useful agent behaviors. These include `DribbleToGoal`, `DashToBall`, `IncorporateObservation`, and others. They are long-term behaviors which should be called on many successive steps. The behaviors themselves tell the user when they are applicable. This ever-growing set of basic behaviors forms the basic building blocks from which students can build their complex agents. The students are free to organize these behaviors, along with any new behaviors they create, in any way they wish.

6 Experiences with Biter

The University of South Carolina has taught a graduate-level course in multiagent systems for several years. The RoboCup soccer simulation problem domain was first adopted for instructional, project-based use for the Fall 1999 semester. Students are divided into groups of two or three, and each group designs and implements a RoboCup team. All groups must write a report on their work and during the final week of classes participate in a class tournament. We made it clear that a group's performance does not directly affect their grade. However, we curiously note that the groups often seemed more motivated by their desire to win the tournament than to achieve a better grade in the class.

In the Fall 1999 semester we gave students a very basic Java client whose only functionality was the ability to parse and exchange messages with the soccerserver. We also made available to them the source code for the CMUnited team [8], authored at Carnegie Mellon and written in 'C'. The CMUnited team had won the pre-

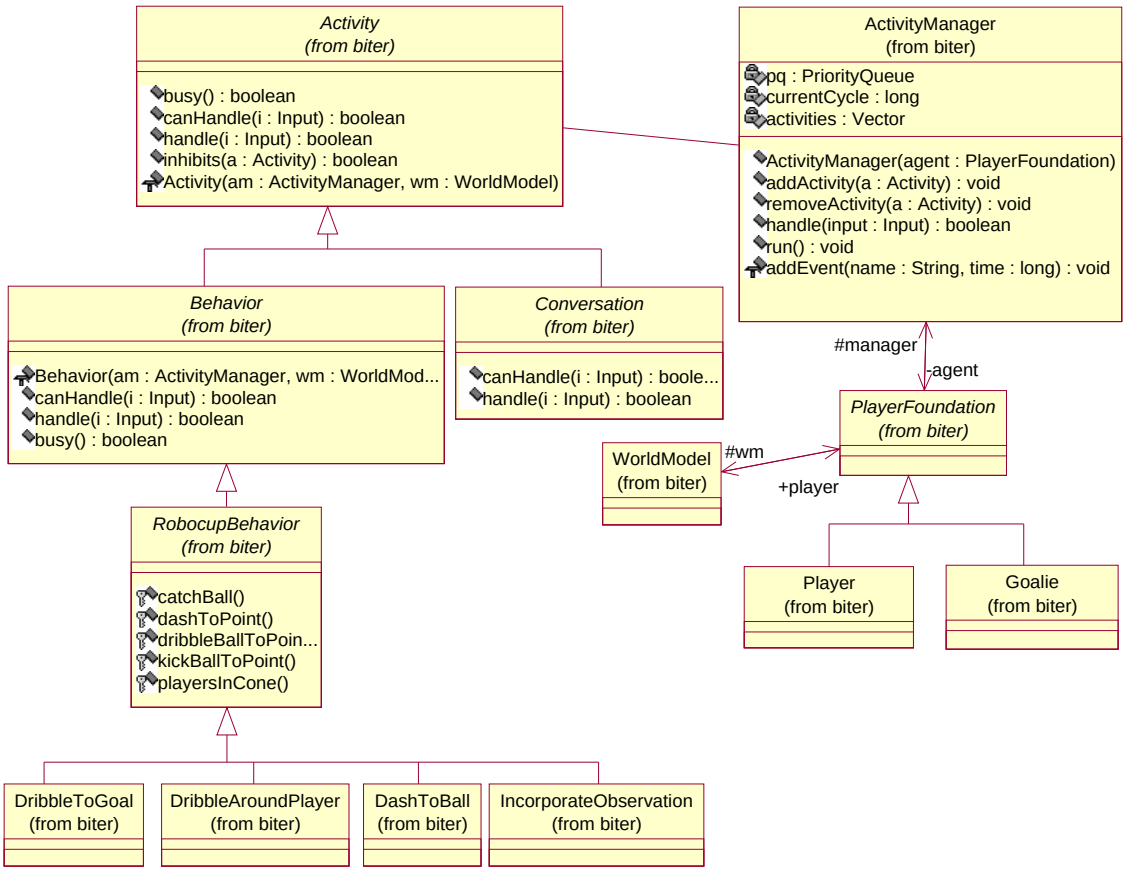

Figure 1: Biter's UML class diagram. We omit many of the operations and attributes for brevity. Italic class names denote abstract classes.

vious two international RoboCup competitions. Due to the complexity of the CMU code, the students unanimously chose to use the simple Java client as their basic framework and to peruse the CMUnited code for ideas.

The final results for the first semester were encouraging. All groups were able to build working teams and participate in the final tournament. Their strategies, however, did not reflect the coordination protocols or behavior selection and planning algorithms we had studied in class. Several groups resorted to a simple "everyone go to the ball and try to kick it towards the goal" strategy. In fact, it was this strategy which won the tournament. A couple of groups implemented very rudimentary "zone" strategies, but these were incomplete. For example, a player dribbling the ball towards the goal would suddenly stop when it reached the end of its zone. Moreover, the code written by many of the groups lacked any structured design and resorted to the use of one large nested if-then-else statement. That is, the students did not do a thorough job at implementing any of the agent architectures described in class.

We believe that part of the reason for this last omission was the lack of well-documented object-oriented designs for agent architectures. Our textbook [10] describes the architectures using very high-level box diagrams. Three quarters into the semester we attempted to remedy this problem by providing the students with a set of UML agent architectures[1] but by then it was too late. The groups were already too committed to their own designs to pay attention to a better alternative.

As a result of these experiences, we developed Biter with the expectation that it would allow the student groups to concentrate more on using the coordination strategies studied in class and help them develop good agent designs.

During the Fall 2000 semester the students were given the version of Biter described in Section 5, with a default behavior of going to the ball and dribbling it towards the goal. The last problem set before the final competition asked the students to implement a team that could beat a team of Biter agents with the default behavior. All groups were able to achieve this goal, with varying degrees of success.

The results of the second tournament were impressive.

[1] Available at `http://www.multiagent.com/arch/`

All of the teams implemented complex strategies. Many of the teams utilized flexible zones, stigmergy [5], and broadcast communications. For example, some groups were able to have the players switch between a set of modes that determined the overall strategy being used (e.g., aggressive versus protective). The players would achieve this without any explicit communication, using only cues from the environment, thereby implementing a form of stigmergy. An example of communication employed by several teams was having a player announce its pass. That is, a player would shout "I am passing to P3" just before making the pass. All the other players, especially P3, that heard the announcement could then behave accordingly.

The quality of the resulting architectures also improved. The player code was no longer a long and hard to understand if-then statement with global mode variables. Instead, the groups encapsulated behavior functionality in the various behavior classes. The behavior to use was chosen based on some pre-determined method such as the priority of the behavior or based on certain features of the current world state. This new modularity also allowed the groups to quickly test new behavior combinations and reject behaviors that actually resulted in worse team performance. We believe that this flexibility contributed to the quality of the final teams.

There were still, however, some areas left for improvement. The teams showed a poor ability to resolve deadlocks. For example, sometimes several players from the same team would all try to take control of the ball at the same time, annulling each other's actions in the process. The teams also never made long passes even when this was clearly the best policy on many occasions. Finally, the real-time performance of the teams was still not up to par with that of competition teams—they were missing action opportunities.

7 Conclusions

Our experiences using RoboCup and Biter have proven to us that these are effective tools for teaching students how to build multiagent systems—encouraging them to develop a distributed mindset. The students often commented how the building of teams was much harder than they initially anticipated—reaffirming our belief that they had not considered the complexities involved in building a multiagent system in a noisy environment. Their final successes, on the other hand, confirmed to us that they had learned how to successfully tackle many of the problems. Our analysis of their code also confirmed that Biter had encouraged most groups to use proper software engineering techniques.

Although our experiences with RoboCup and Biter have been at the graduate level, we fully expect that they will be useful tools for undergraduate education. The STEELMAN draft of the Computing Curricula 2001 (CC2001) recognizes that distributed systems topics need to be introduced with more rigor in an undergraduate CSE education. Topics related to distributed systems are present in each of the following CS body of knowledge core areas, as defined in CC2001: Algorithms and Complexity, Architecture and Organization, Operating Systems, Intelligent Systems, and Net-Centric Computing [3]. Each of these core areas is further subdivided into topics and units. The STEELMAN draft of CC2001 presents sample curricular components that demonstrate possible strategies for integrating topic and unit coverage into an undergraduate CSE educational experience. Our work couples topically with the proposed intermediate course CS240 - Intelligent Systems. The RoboCup simulation league, with the aid of the Biter framework, could easily serve as a project-based component for this Intelligent Systems course.

Biter continues to evolve. New features and behaviors are being added and we expect the pace to quicken as more users start to employ it for pedagogical and research purposes.

References

[1] Biter: A robocup client. `http://source.cse.sc.edu/biter/`.

[2] Robocup initiative. `http://www.robocup.org`.

[3] Computing curricula 2001, steelman draft, August 2001. `http://www.computer.org/education/cc2001/steelman/cc2001`.

[4] Jennings, N. R. On agent-based software engineering. *Artificial Intelligence 117* (2000), 277–296.

[5] Kube, C. R., and Bonabeau, E. Cooperative transport by ants and robots. Santa Fe 99-01-008.

[6] Oram, A., Ed. *Peer-to-Peer*. O'Reilly, 2001.

[7] Resnick, M. *Turtles, Termites and Traffic Jams*. The MIT Press, 1994.

[8] Stone, P. *Layered Learning in Multiagent Systems: A Winning Approach to Robotic Soccer*. The MIT Press, 2000.

[9] Vidal, J. M., Buhler, P. A., and Huhns, M. N. Inside an agent. *IEEE Internet Computing 5*, 1 (January-February 2001).

[10] Weiss, G., Ed. *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*. MIT Press, 1999.