Teaching Multiagent Systems using RoboCup and Biter

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 describe 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 three 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 was held in Fukuoka, Japan in cooperation with Busan, Korea in June, 2002, scheduled coincide with the "2002 World Cup Korea/Japan." The simulation league had 46 teams with 165 participants from 15 countries. The whole tournament boasted 1022 participants from 30 countries.

Interactive Multimedia Electronic Journal of Computer-Enhanced Learning, 4(2), 2002.

This paper presents our experiences using RoboCup and Biter—our agent framework—to teach three 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. Having students 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.

2 The Distributed Mindset

In [11], Resnick shows how humans have a tendency to erroneously assume that there is centralized control in 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 computer science and engineering 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 [10].

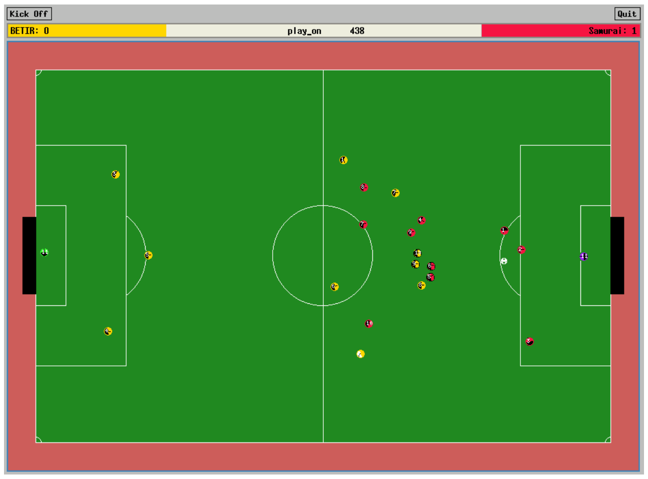

Figure 1: The soccerserver display.

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 interprocess 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 its 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 in the direction that 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 applied. In the case of the dash command the server updates the player's position by taking into account its current momentum and force. The *turn* command turns the player in the appropriate direction. 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 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 shout any arbitrary string which will be heard by some of the nearby players.

4 Using RoboCup for Teaching

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) [14] 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 [12]. 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. A player receives sensory input, relative to his current position, consisting of vectors that point to the static and dynamic objects in his field of view. Since static objects have fixed locations, they are important in calculating a player's absolute position on the field of play. If a player knows his absolute location, the relative positions of the dynamic objects in the sensory input can be transformed into absolute locations.

As sensory information about dynamic objects is placed into Biter's world model it is time stamped and the world model is updated. We first calculate the player's absolute position using some of the closest static objects as guide. We then use the player's position to calculate the absolute position of all dynamic objects. All information is discarded after its age exceeds the user-defined limit. Users can experiment with this limit. A small value leads to a purely reactive agent, a large value leads to the agent seeing "ghosts" of players that are not there anymore.

Access to world model data should be simple; however, approaching this extraction problem too simplistically leads to undesirable cluttering of code. This code obfuscation occurs with access strategies that litter loop and test logic within every routine that accesses the world model. Biter utilizes a decorator pattern [5] which is used to augment the capabilities of Java's ArrayList iterator. The underlying technique used is that of a filtering iterator. This filtering iterator traverses another iterator, only returning objects that satisfy a given criteria. Biter utilizes regular expressions for the selection criteria. For example, depending on proximity, the soccer ball's identity is sometimes reported as "ball" and other times as "Ball". If our processing algorithm calls for the retrieval of the soccer ball from the world model, we would initialize the filtering iterator with the criteria [bB]all to reliably locate the object. Since the filtering criterion is regular expression based, we are able to construct powerful extraction routines without incurring the complexity of coding error-prone compound conditionals.

Although access to the world model has been streamlined, creating more concise and algorithm-revealing code, it remains difficult to fully understand the behavior of the players. At times it seems the only way to understand moments of unexplainable behavior is to have access to the player's world model. Dumping the contents of the world model to a file for later interpretation is unnecessarily complex and unwieldy. To attack this problem, Biter provides a run-time visual display of a player's internal view of his environment. When a Biter agent is started, a command-line parameter is used to enable the graphical display of the world model. The display is served by an independent thread and utilizes double buffering for smooth animation. The overhead view of the field shows all static objects and the dynamic objects currently found in the player's world model. Whenever stale elements are encountered, an algorithm is run which merges its display color with the background color of the field. Visually, this has the effect of having stale elements fade away as they age. The graphical display of a player's world model can be compared to the soccer monitor's display for purposes of independent verification and validation of the player's world model contents. This powerful debugging feature has saved users countless hours of fruitless troubleshooting and helps them focus on other multiagent system implementation issues.

5.2 The Generic Agent Architecture

Practitioners new to agent-oriented software engineering [7] 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 [15, 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 2.

An agent is propelled to act only after receiving some form of input. That is, after the activity manager receives a new object of the `Input` class. This class has three sub-classes: `SensorInput`, `Message`, and `Event`. A `SensorInput` is a set of inputs that come directly from the agent's sensors. Biter provides a parsing function that transforms the input from its original format—a list—into an object of this class. In most implementations a class hierarchy should be created under this class in order to differentiate between the various types of sensor inputs. Biter defines `ObjectInfo` and `ObjectInfoContainer` as extensions of this class. The `Message` class represents a message from another agent. Robocup players can use a a broadcast mechanism ("say") to send messages to all nearby players. Finally, the `Event` class is a special form of input that represents an event the agent itself created. Events function as alarms set to go off at a certain time. They are important because they provide a way to implement timeouts. Timeouts are used when waiting for a reply to a message, when waiting for some input to arrive, or when repeatedly taking an action in the hope of generating some effect.

Biter implements a special instance of `Event` which we call the `act` event. This event fires when the time window for sending an action to the soccer server opens. That is, it tries to fire every 100ms, in accordance with the soccerserver's main loop. Since the messages between Biter and the soccerserver can be delayed, and their clocks can get skewed over time, the actual firing time of the `act` event needs to be constantly monitored. Biter uses an algorithm similar to the one used in [12] for keeping these events synchronized with the soccerserver.

The `Activity` class represents our basic building block. Biter agents are defined by creating a number of activities and letting the activity manager schedule them as needed. The `Activity` class has three main member functions: `canHandle`, `handle`, and `inhibits`.

The `canHandle` member function receives an input object as an argument and returns `true` if the activity can handle the input, that is, if it can execute as a consequence of receiving that input. This function could not only consider the contents of the input, but it could also consider the agent's current internal state, the agent's world model, etc. Since this is a generic framework, we do not constrain the `canHandle` function to only access a certain subset of the available data. That decision is left to the software engineer who wants to refine the architecture. The only requirement we make is for the function to be speedy since it will need to be called after

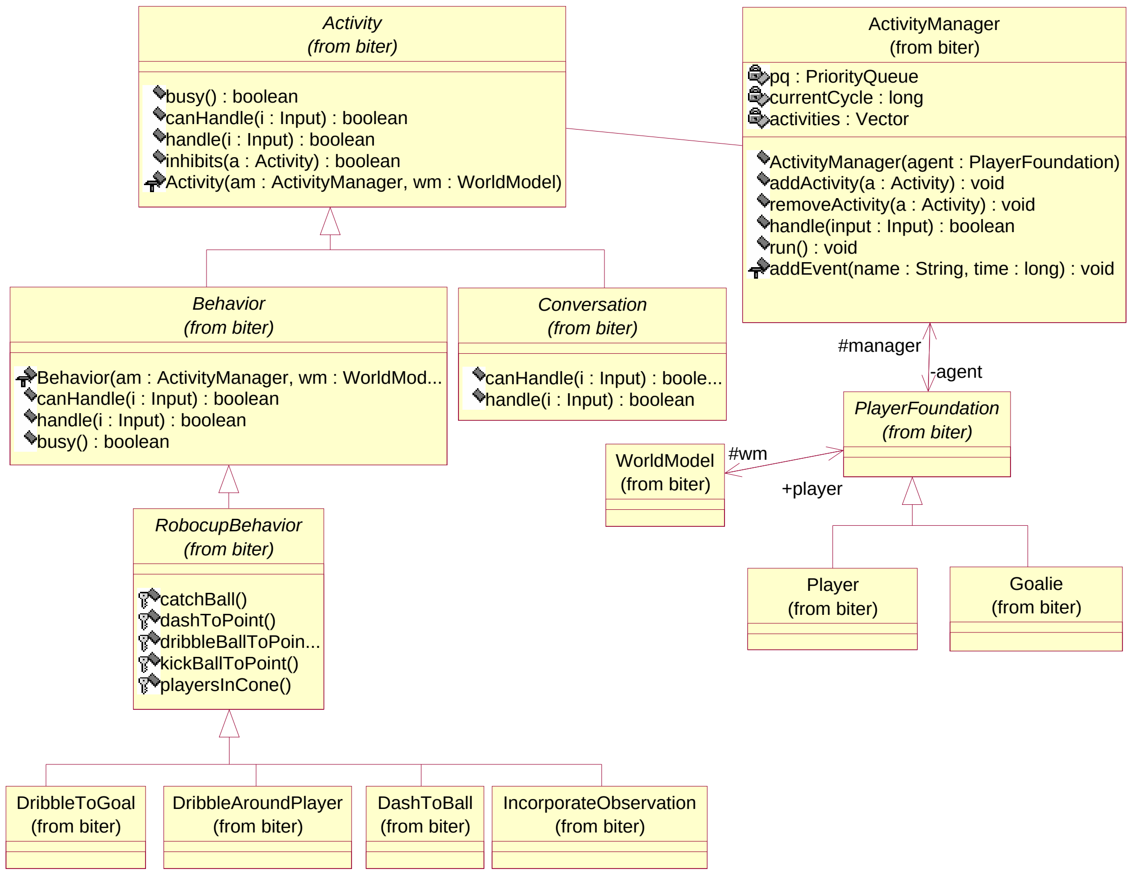

Figure 2: Biter's UML class diagram. We omit many of the operations and attributes for brevity. Italic class names denote abstract classes.

each new input has arrived.

The `handle` member function is called when the activity is chosen to handle that input. It is called when the activity manager wants the activity to execute with the given input. This function usually generates one or more atomic actions, sets some member variables, and returns. A call to the `handle` function executes the next step in the activity, the step that corresponds to the received input. The function can set member variables as a way to maintain a state between successive invocations. This state allows the activity to implement multi-step plans and other complex long-term behaviors. The `handle` function will return `true` when the activity is done, at which point it will be deleted. We expect that in most agents there will be a set of persistent activities that are never done and always return `false`.

Finally, the `inhibits` member function receives an `Activity` object as a parameter and returns `true` if that activity is inhibited by the current one. This function implements the control knowledge which the activity manager will use to determine which activity to execute. The use of this function mirrors the use of subsuming behaviors in the subsumption architecture [4]. However, the function can also consult state variables in order to calculate its value, thereby extending the functionality. Since the activities are organized in a hierarchy, this function is able to easily inhibit whole subtrees of that hierarchy. This allows users to add new activities without having to modify all existing ones.

A significant advantage of representing each activity by its own class, and with the required member functions, is that we enforce a clear separation between the behavior knowledge and the control knowledge. That is, the `handle` function implements the knowledge about how to accomplish certain tasks or goals. The `canHandle` function tells us under which conditions this activity represents a suitable solution. Meanwhile the `inhibits` function incorporates some control knowledge that tells us when this activity should be executed. This separation is a necessary requirement of a modular and easily expandable agent architecture.

The `Behavior` class is an abstract class that groups all long-term behaviors of the agent. We define these behaviors as series of atomic actions. For example, a robotic behavior might be to "avoid obstacles", while a software agent might have a "gather data from sources" behavior. Behaviors can, like all activities, create new activities and add them to the set of activities.

Biter defines its own behavior hierarchy by extending the `Behavior` class, starting with the abstract class `RobocupBehavior` which implements many useful functions. The hierarchy continues with standard behaviors such as `DashToBall`, `IncorporateObservation`, and `DribbleToGoal`. For example, a basic Biter agent can be created by simply adding these three behaviors to a player's activity manager. The resulting player would always run to the ball and then dribble it towards the goal.

The `Conversation` class is an abstract class that serves as the base class for all the agent's conversations. In general, we define a conversation as a set of messages sent between one agent and other agents for the purpose of achieving some goal, e.g, the purchase of an item, the delegation of a task, etc. A GAA implementation defines its own set of conversations as classes that inherit from the general `Conversation` class. For example, if an agent wanted to use the contract-net protocol, it would implement a contract-net class that inherits from `Conversation`.

Conversations implement protocols. Most protocols can be represented with a finite state machine where the states represent the current status of the conversation and the edges represent the messages sent between agents (see [9] for specific proposal that extends UML to cover agent conversations). In some protocols each agent will play one of the available "roles." For example, in the contract-net protocol agents can play the role of contractor or contractee. The conversations will, therefore, implement a finite state machine.

Multiple conversations can be handled by having the existing conversation add a new one to the set of activities. For example, if a message that starts a new conversation (e.g., a request-for-bids) is received by an agent the `canHandle` function of the appropriate conversation will return `true` even if the conversation is already busy, that is, even if it is not in its starting state. When the `handle` function is called with the new message the conversation will recognize that its busy and create a new conversation, add it to the action manager, call the new conversation's `handle` method with the new input, and return. In this way, a new conversation object is created to handle the new message. Behaviors can use the same method to initialize a conversation.

For example, a "move to point" behavior might realize that another agent is blocking the path and start a conversation with that agent in an effort to convince it to move out of the way. If only one instance of a conversation is desired the user can implement a conversation factory [5, Factory Method] in order to dynamically limit the number and type of conversations.

```
input = the new input
activities = set of all activities
matches = new Vector()
for all i in activities do
  if i.canHandle(input) then
    matches.addElement(i)
  end if
end for
uninhibited = new Vector()
for all i in matches do
  inhibited = false
  for all $j \neq i$ in matches do
    if j.inhibits(a) then
      inhibited = true
    end if
    if not inhibited then
      uninhibited.addElement(i)
    end if
  end for
end for
chosen = uninhibited.choseRandom()
if chosen.handle() then
  removeActivity(chosen)
end if
```

Figure 3: ActivityManager.handle(Input i)

We are also planning to add error handling and verification functions to the top-level `Conversation`. Specifically, many conversations will want to implement some timeout mechanism for expected replies, as well as a method for determining what the next action should be (e.g., resend the message, send another message, fail). Given the commonality of this functionality, it makes sense for us to implement it on the base class.

The `ActivityManager` picks one of the activities to execute for each input the agent receives. It implements the agent's control loop. The manager runs in its own thread, where it receives input from the sensors and dispatches it to the appropriate activity. The dispatching is done by the `handle` function, shown in Figure 3, which determines which of the activities will actually handle the input. The algorithm it implements echoes the type of control mechanism implemented by subsumption and Belief, Desire, Intention [6] architectures. The function first finds all activities that can handle the input; from this group it chooses one which is not inhibited by any other one in the group and asks it to handle the input. Since the inhibition function can be arbitrarily defined by its activity, the ordering becomes very flexible. That is, the user of the GAA has options ranging from no organization (no activity inhibits any other activity), to a static organization (activities inhibit a fixed type of activities), to a dynamic organization (activities inhibit based on many other factors). As an agent matures, the user can choose to increase the organizational complexity without re-implementing the architecture.

All agent implementations that extend Biter must follow a series of steps. First the agent must instantiate an activity manager object. The agent then adds the desired activities to this object. These are the activities that define its overall behavior. It then calls the `run` method on the activity manager in order to start it running. At this point the manager takes complete control and enter its infinite loop, choosing which behavior to execute every time. In general, the user should not need to modify the manager. All the control knowledge is stored in the `canHandle` and `inhibits` methods which are defined by the user.

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 Spring 2000 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 Spring 2000 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 [12], authored at Carnegie Mellon and written in 'C'. The CMUnited team had won the previous 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 [15] 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. All of the teams implemented complex strategies. Many of the teams utilized flexible zones, stigmergy [8], 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

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

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.

In the Fall of 2001 class we again used the same Biter code, this time with a few new low-level behaviors and bug-fixes. Once again we noticed improvements in the team performance, albeit not as drastic as the previous year. All the teams exhibited true team behavior. They engaged in a lot of passing and, on occasion, had pass sequences that involved three players—from first, to second, to third, to the goal. Overall the students' experiences continued to be positive.

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. In general, the strategies were brittle in that they would stop working even after small changes were made to the soccerserver parameters. Finally, the real-time performance of the teams was still not up to par with that of competition teams—they were missing action and coordination 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. After three years of using Robocup to teach multiagent systems we feels that we have learned a few valuable lessons.

- Robocup is an excellent motivator. The quality of the students' final projects was better than on other similar projects where the only reward was a good grade.
- The best performing teams on the tournament always have better designs and are built by the better programmers. That is, we have never seen a badly designed team win or do well on the tournament.
- Students invariably underestimate the difficulty of achieving coordinated team behavior. We surmise that this ignorance contributes to their early eagerness to start building a team.
- Even the best teams are brittle—small changes in the game parameters (e.g., wind speed, noise, etc.) break them. This is to be expected since engineering robustness remains an open research problem.
- There is an obsession with response time, even when it is already good enough. Many groups would spend time optimizing the speed of the Biter code even if this did not actually change the performance of their team in any way.[2] We believe that this behavior arises because the students know how to make programs faster, so they focus on applying those techniques instead of focusing on achieving coordination, which is the only way to improve their team performance.

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. We are currently working on the addition of a GUI for the visual development of agents using Java Beans, as well as more low-level behaviors. We envision a system which will allow users to draw graphs with the basic behaviors as the vertices and "inhibits" links as the directed edges. These edges could be annotated with some code. Our system would then generate the Java code that implements the agent. That is, the behaviors we have defined can be seen as components [13] which the programmer can wire together to form aggregate behaviors. This system will allow inexperienced users to experiment with multiagent systems'

[2]Since the time slice is 100ms, sending responses sooner than that does not achieve anything.

design, both at the agent and the multi-agent levels. We also believe the system will prove to be useful to experienced multiagent researchers because it will allow them to quickly prototype and test new coordination algorithms.

References

[1] Biter: A robocup client. `http://jmvidal.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] Brooks, R. A. Intelligence without representation. *Artificial Intelligence 47* (1991), 139–159.

[5] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[6] Georgeff, M., Pell, B., Pollack, M., Tambe, M., and Wooldridge, M. The Belief-Desire-Intention model of agency. In *Proceedings of Agents, Theories, Architectures, and Languages* (1999).

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

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

[9] Odell, J., Parunak, H. V. D., and Bauer, B. Representing agent interaction protocols in UML. In *Proceedings of the Fourth International Conference on Autonomous Agents* (2000).

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

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

[12] Stone, P. *Layered Learning in Multiagent Systems*. MIT Press, 2000.

[13] Szypersky, C. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1999.

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

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