# Biter: A Platform for the Teaching and Research of Multiagent Systems' Design using RoboCup

Paul Buhler[1] and José M. Vidal[2]

College of Charleston, Computer Science, 66 George Street, Charleston, SC 29424
pbuhler@cs.cofc.edu,
University of South Carolina, Computer Science and Engineering,
Columbia, SC 29208
vidal@sc.edu

**Abstract.** We introduce Biter, a platform for the teaching and research of multiagent systems' design. Biter implements a client for the RoboCup simulator. It provides users with the basic functionality needed to start designing sophisticated RoboCup teams. Some of its features include a world model with absolute coordinates, a graphical debugging tool, a set of utility functions, and a Generic Agent Architecture (GAA) with some basic behaviors such as "dribble ball to goal" and "dash to ball". The GAA incorporates an elegant object-oriented design meant to handle the type of activities typical for an agent in a multiagent system. These activities include reactive responses, long-term behaviors, and conversations with other agents. We also discuss our initial experiences using Biter as a pedagogical tool for teaching multiagent systems' design.

## 1 Introduction

The field of multiagent systems traces its historical roots to a broad array of specialties and disciplines in the fields of AI, logics, cognitive and social sciences, among others. Within the academic setting, pedagogical approaches are needed that provide opportunities for students to perform meaningful experimentation through which they can learn many of the guiding principles of multiagent systems development. The Biter framework was designed to enable a project-based curricular component that facilitates the use of the RoboCup simulator within the classroom setting.

The RoboCup simulator has many qualities that make it an excellent testbed for multiagent systems' research and for teaching multiagent systems' design. First, the simulator presents a complex, distributed, and noisy environment. Second, in order to win a game, it is necessary to foster cooperation and coordination among the autonomous agents that compose a team. Third, students are engaged by the competitive aspect of RoboCup and many are highly motivated by the prospect of defeating their classmates in a game of simulated soccer. Finally, the RoboCup initiative has generated a wealth of research materials that are easily located and consumed by students.

While all these characteristics make the RoboCup simulator a great platform, there are several aspects which make it difficult for a beginner researcher to use productively.

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. The agents must either be built from scratch or by borrowing code from one of the existing RoboCup tournament teams. These teams are implemented for competition, not to be used as pedagogical tools. Their code is often complex, documentation scarce and they can be hard to decipher.

The Biter system addresses each of these issues in an effort to provide a powerful yet malleable framework for the research and study of multiagent systems.

## 2 The Biter Platform

Biter provides its users with an 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) 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. Its source code, Javadoc API, and UML diagrams are available at `http://source.cse.sc.edu/~biter/`.

### 2.1 Biter's World Model

In the RoboCup domain it has become clear that agents need to build a world model [2]. This world model should contain slots for both static and dynamic objects. Static objects have a field placement that does not change during the

course of a game. Static objects include flags, lines, and the goals. In contrast, dynamic objects move about the field during the game; they represent 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 positions, they can be used to calculate a player's absolute location on the field of play. The absolute location of a player is used to transform the relative positions of the dynamic objects into absolute locations.

As sensory information about dynamic objects is placed into Biter's world model it is time stamped. World model data is discarded after its age exceeds a user-defined limit. Users can experiment with this limit. Small values lead to a purely reactive agent, while larger values retain a history of environmental change.

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 [1] 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. Accessing the world model elements, with the aid of a filtering iterator, has helped to reduce the overall complexity of student-authored code. Simplification of the interface between the student's code and the world model, allows students to focus more directly on building behavior selection and planning algorithms.

### 2.2 The Generic Agent Architecture

Practitioner's new to agent-oriented software engineering often stumble when building an agent that needs both reactive and long-term behaviors, often 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. Biter implements a GAA [3] which provides the structure needed to guide users in the development of a solid object-oriented agent architecture.

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 behaviors and conversations. Behaviors are actions taken over a series of time steps. Conversations are series of messages exchanged between

agents. 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.

The `Activity` abstract class represents our basic building block. Biter agents include a collection of activities that the activity manager schedules as needed. A significant advantage of representing each activity by its own class is that we enforce a clear separation between behavior and control knowledge. This separation is a necessary requirement of a modular and easily expandable agent architecture.

The Activity class has three main member functions: `handle`, `canHandle`, and `inhibits`. 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.

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 will 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.

The `ActivityManager` picks one of the activities to execute for each input the agent receives. That is, an agent is propelled to act only after receiving a new object of the `Input` class. The Input class has three sub-classes: `SensorInput`, `Message`, and `Event`. A `SensorInput` is a set of inputs that come directly from the agent's sensors. The `Message` class represents a message from another agent. That is, we assume that the agent has an explicit communications channel with the other agents and the messages it receives from them can be distinguished from other sensor input. 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.

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 their clocks can get skewed over time; therefore, the actual firing time of the `act` event needs to be constantly monitored. Biter uses an algorithm similar to the one used in [2] for keeping these events synchronized with the soccerserver.

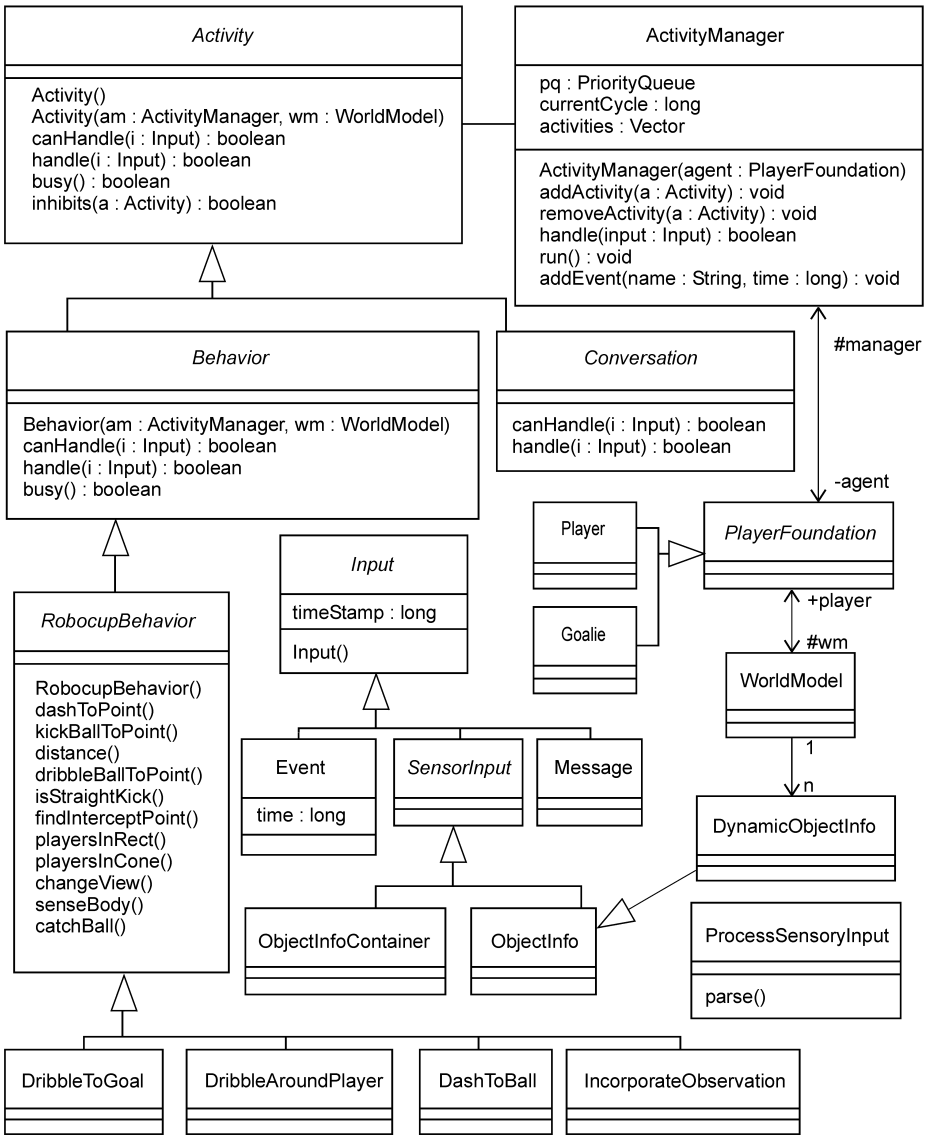


**Fig. 1.** Biter's UML class diagram. We omit many of the operations and attributes for brevity. Italic class names denote abstract classes.

## 3 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 has been adopted for instructional, project-based use. Without the Biter framework, students spent the majority of their time writing support code that could act as scaffolding from which they could build a team of player agents. Multiagent systems theory and practice took a backseat to this required foundational software construction. At the end of the semester, some teams competed, however the majority of these were reactive agents due in part to the complexity of creating and maintaining a world model.

With Biter available for student use, the focus of development activities has been behavior selection and planning. The GAA allows students to have hands-on experience with both reactive and BDI architectures. Students are no longer focused on the development of low-level skills and behaviors, but rather on applying the breadth and depth of their newly acquired multiagent systems knowledge. Biter provides a platform for flexible experimentation with various agent architectures.

## 4 Further Work

Biter continues to evolve; new features and behaviors are added continuously. We expect the pace to quicken as more users start to employ it for pedagogical and research purposes. One of our plans is the addition of a GUI for the visual development of agents. 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 which the programmer can wire together to form aggregate behaviors. This system will allow inexperienced users to experiment with multiagent systems' 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. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
2. Peter Stone. *Layered Learning in Multiagent Systems: A Winning Approach to Robotic Soccer*. The MIT Press, 2000.
3. José M. Vidal, Paul A. Buhler, and Michael N. Huhns. Inside an agent. *IEEE Internet Computing*, 5(1), January-February 2001.