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

Paul Buhler[1] and José M. Vidal[2]

[1] College of Charleston, Computer Science, 66 George Street, Charleston, SC 29424
pbuhler@cs.cofc.edu,
[2] 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 detail our experiences using Biter as a pedagogical tool for teaching multiagent systems' design.

1 Introduction

The Robocup tournament has proven to be successful at bringing together researchers from different areas, such as robotics, AI, multiagent systems, machine vision, etc., and getting them involved in solving a single problem. That success should be applauded. However, we believe that the simulated Robocup league can also prove to be an excellent platform for the study and the teaching of multiagent systems' design, as long as a suitable agent architecture is provided. This article introduces such an architecture.

The Robocup simulator has many qualities which make it an excellent testbed for multiagent systems' research and for teaching 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. Also, the international Robocup tournament inspires researches and shows them what is possible.

While all these characteristics make Robocup a great platform, there are some aspects which make it hard for a beginner researcher to use it for multiagent research.

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 teams. Researchers new to agent design need some guidance in establishing a basic agent architecture and generally prefer not to spend time understanding the often complex designs of existing Robocup teams. Robocup teams are designed to win, not to be used as pedagogical tools, so their code can often be unwieldly to a beginner.

The Biter system addresses all these issues in an effort to provide a powerful yet malleable framework for the research and study of multiagent systems. Specifically, Biter was designed to enable a project-based curricular component that facilitates the use of RoboCup within the classroom setting. Also, by implementing a generic agent architecture, Biter allows users to explore both strong and weak notions of agency [9].

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 [1].

2.1 Biter's World Model

In the RoboCup domain it has become clear that agents need to build a world model [6]. 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 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.

The Biter framework provides a world model that contains both static and dynamic objects. 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.

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 [3] 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 players 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.

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. 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 [7] 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 [8, 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.

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 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. 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. This is possible in the Robocup domain, but there might be other domains where it is not trivial to distinguish a message from just a feature of the environment as, for example, if the agents communicate by moving objects on a landscape. In these cases the use of messages and, as a consequence, conversations is not recommended.

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 [6] 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, or 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 each new input has arrived.

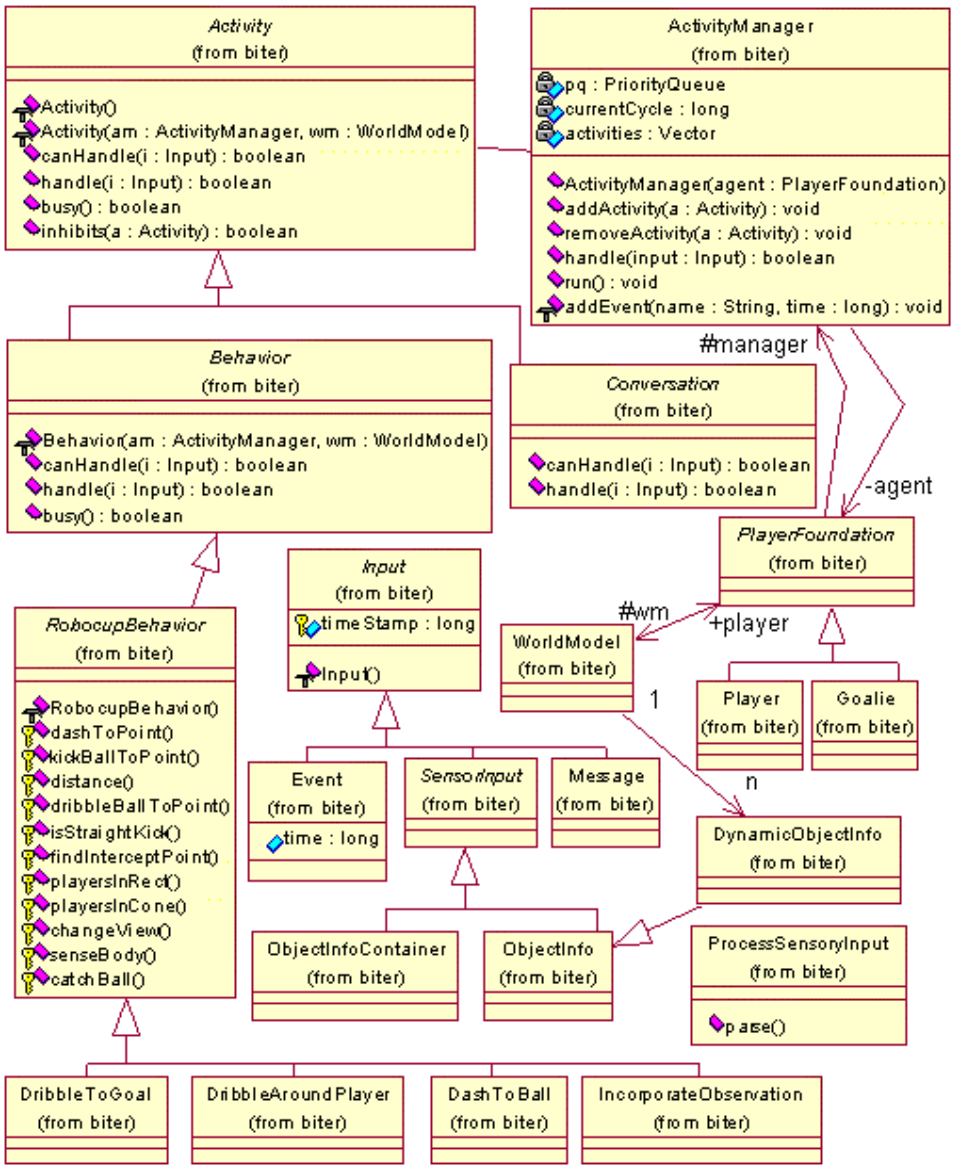

Fig. 1. Biter's UML class diagram. We omit many of the operations and attributes for brevity. Italic class names denote abstract classes.

The `handle` member function is called when the activity is chosen to handle that input, calling it means the activity manager want 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.

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 [2]. 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 [5] 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 [3, Factory Method] in order to dynamically limit the number and type of conversations.

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 2, 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 BDI [4] 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.

```
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 ≠ 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
```

Fig. 2. ActivityManager.handle(Input i)

3 Experiences with Biter

Our University 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 for the past two semesters. During the first semester, 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, teams competed, however the majority were reactive agents due in part to the complexity of creating and maintaining a world model. The Biter framework was an outgrowth from this experience.

With Biter available for student use, the focus of team development 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 Summary and Further Work

We have introduced Biter, a platform that has enabled the teaching of multiagent systems' design using Robocup. Biter does all the needed bookkeeping tasks such as communicating with the soccerserver and maintaining a world model, as well as providing many basic behaviors such as dribbling, and interception. Biter also implements a generic agent architecture which provides a framework for the development of complex agents. We believe Biter will encourage further research of multiagent systems' design because it provides a solid object-oriented agent framework which separates control knowledge from domain knowledge and which is easy to extend.

Biter continues to evolve. New features and behaviors are added constantly. We expect the pace to quicken as more users start to employ it for pedagological 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 expect to use the system in undergraduate artificial intelligence classes. 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://source.cse.sc.edu/biter/`.
2. Rodney A. Brooks. Intelligence without representation. *Artificial Intelligence*, 47:139–159, 1991.
3. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
4. Michael Georgeff, Barney Pell, Martha Pollack, Milind Tambe, and Michael Wooldridge. The Belief-Desire-Intention model of agency. In *Proceedings of Agents, Theories, Architectures, and Languages*, 1999.
5. James Odell, H. Van Dyke Parunak, and Bernhard Bauer. Representing agent interaction protocols in UML. In *Proceedings of the Fourth International Conference on Autonomous Agents*, 2000.
6. Peter Stone. *Layered Learning in Multiagent Systems: A Winning Approach to Robotic Soccer*. The MIT Press, 2000.
7. José M. Vidal, Paul A. Buhler, and Michael N. Huhns. Inside an agent. *IEEE Internet Computing*, 5(1), January/February 2001.
8. Gerhard Weiß, editor. *Multiagent Systems : A Modern Approach to Distributed Artificial Intelligence*. The MIT Press, 1999.
9. Michael Wooldridge and Nick R. Jennings. Intelligent agents: Theory and practice. *Knowledge Engineering Review*, 10(2), 1995.