A Generic Agent Architecture for Multiagent Systems

José M. Vidal
Swearingen Engineering Center
University of South Carolina
Columbia, SC 29208
vidal@sc.edu
http://jmvidal.ece.sc.edu

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

ABSTRACT

We introduce the Generic Agent Architecture (GAA) along with Biter—an implementation of our GAA for the RoboCup domain. 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 show how small modifications in the GAA implementation can lead to a subsumption agent or to a BDI agent. Finally, we present our Biter implementation as a proof of concept and use it to illustrate the added functionality that a user of the GAA must implement in a specific domain in order to utilize our GAA.

1. INTRODUCTION

When building an agent that will be part of a multiagent system one is faced with a challenging set of obstacles. The agent must be able to initiate and carry out conversations with other agents, to carry out long-term behaviors, to take immediate actions when necessary, and to handle error conditions such as missing messages or noisy input. Furthermore, the agent implementation should follow the standard software engineering practices of data abstraction and specification. A good design allows one to easily distribute implementation responsibilities among developers and to quickly add new functionalities at some later date.

For example, imagine that we are building a team of software agents that will participate in a simulated soccer tournament, such as the Robocup challenge. We decide to implement a coordination protocol which will sometimes require the players to send and receive several messages in a row, maintaining a conversation. Our players must also implement long-term behaviors, such as dribbling the ball and finding an open teammate, which consist of several atomic actions. Finally, we want an implementation that will make it easy to add new functionality as we learn more about the problem domain. This set of requirements is common to many multiagent domains.

USC CSCE TR-2002-011

Faced with this set of requirements, it becomes clear that an agent architecture is needed to structure the programming task. We can choose from a number of agent architectures. The subsumption architecture [7] and the Belief Desires Intentions (BDI) architecture [6] are among the most popular.

The subsumption architecture is a reactive architecture which, if implemented so as to faithfully follow the original design, is completely reactive. That is, it does not maintain any state. It supports the use of many behaviors and provides a way to specify how behaviors inhibit each other. These inhibition relationships are used to determine which behavior is executed at each time.

The BDI architecture provides a more complex control mechanism. A BDI agent has a set of desires and a set of plans associated with each desire. The agent also has a set of beliefs which include everything it knows about the state of the world and the agents in it, as well as the agent's internal state. The BDI control mechanism then chooses which of the plans to intend by first finding the current desires, then the set of plans that satisfy these desires, and then choosing one of them. Once a plan is intended the BDI control mechanism has some rules to determine how long it will stay intended. A few of the many BDI implementations include the Procedural Reasoning System (PRS) [14], the University of Michigan PRS (UM-PRS) system [17], and dMARS [10]. Each one implements different ways of choosing which plan to execute, some use fixed priorities while others use meta-reasoning or other methods.

These architectures provide us with a solid foundation for building an agent. But, there are still several major stumbling blocks that one faces when trying to implement an agent, for a multiagent system, using an object-oriented language.

1. We need a way to easily implement conversations and long-term behaviors. BDI plans provide some support for this but they are not explicit enough.

2. We need a detailed object-oriented design to guide our implementation. Subsumption implementations are usually hardware-based, while BDI implementations are usually based on logic and rule-based programming [12], or implement a new language and its interpreter—as done by UM-PRS. In general, it is very hard for novices to translate the subsumption and BDI descriptions given in textbooks (e.g., [25]) into a coherent object-oriented implementation.

3. We need a structured way of dealing with error conditions such as lost messages and lost sensor readings.

4. We might not wish to deal with the complexities of a BDI control system, while a purely reactive system might not be enough for our needs. Also, as our agent matures the requirements might change. Therefore, we need an architecture that can be modified to act as purely reactive, purely goal-driven, or anywhere in between.

In this article we will present our Generic Agent Architecture (GAA) which provides support for conversations, long-term behaviors, and reactive behaviors, all in a modular way which separates behavior knowledge from control knowledge. That is, it separates the code that tells the agent how to do something, from the code that tells it when to apply each behavior. The GAA is meant as a general design for those who are new to agent-oriented software engineering and who wish to implement their whole system from scratch. Those who do not want to build their own system can instead use one of the many existing agent systems (see Section 6).

Section 2 gives a summary of the architecture. Section 3 shows how that GAA can be used to implement a subsumption architecture, while Section 4 shows how to implement a BDI architecture. Section 5 introduces Biter—an implementation of the GAA for the Robocup domain, and describes a typical use of the GAA for a specific problem. Finally, Section 6 presents some related work and Section 7 summarizes our contribution.

2. THE GENERIC AGENT ARCHITECTURE

The GAA [24] provides a general design for building agents, using an object-oriented language, which will participate in a multiagent system. Specifically, the agents are assumed to 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 way to model autonomous agents [25, Chapter 1] and captures many agent applications. We also assume the environment is non-deterministic and the agent can have near real-time requirements. For example, the agent might be required to take an action within a certain time window.

The GAA provides a mechanism for scheduling activities each time the agent receives some form of input. The various input types are described in Section 2.1. An activity, described in Section 2.2, 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 set of time steps. The ActivityManager, presented in Section 2.3, determines which activity should be called to handle any new input.

2.1 Input

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`. Their relationships are shown by the UML [5] class diagram in Figure 1.

A `SensorInput` is a set of inputs that come directly from the agent's sensor. We generally assume there exists 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.

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 almost all domains, but there might be cases 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 are like 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.

2.2 Activities

The `Activity` class represents our basic building block. A GAA agent is 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 relationships between activities and the activity manager is shown by the UML class diagram on Figure 2.

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 our 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.

The **`handle`** member function is called when the activity is chosen to handle that input. It gets called when the activity manager wants it 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 most agents will have a set of persistent activities that are never be 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

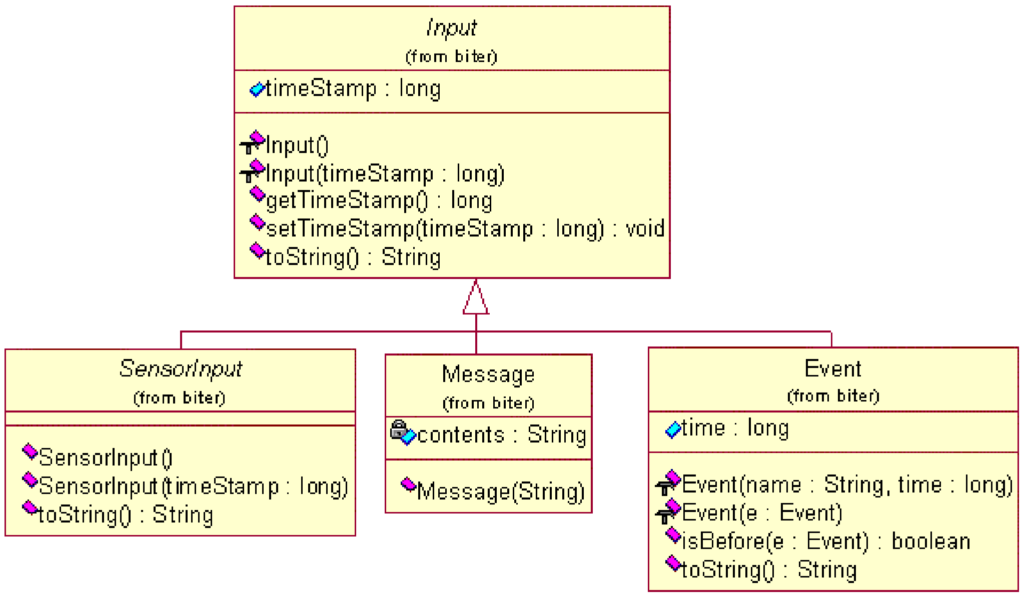

Figure 1: UML class diagram of the Input hierarchy.

mirrors the use of subsuming behaviors in the subsumption architecture. 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 would allow us 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 easy to expand agent architecture.

2.2.1 Behavior

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.

We expect that users will define their own behavior hierarchy under this class, starting with a general class and defining progressively more specialized behaviors. The general behavior class will typically include a few utility functions that have proven useful for the particular problem domain. The subclasses of this class will serve as a way of grouping similar behaviors. For example, in the robotic soccer domain we might have a general abstract "soccer" behavior, followed by a "dribble" behavior, followed by a "dribble when near an opponent" behavior. The "soccer" behavior would implement the utility functions, while the "dribble" behavior would implement an actual physical behavior which "dribble when near an opponent" specializes.

2.2.2 Conversation

The `Conversation` class is an abstract class that serves as the base class for all the agent's conversations. We define a conversation as a set of messages sent between the 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 [21] 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

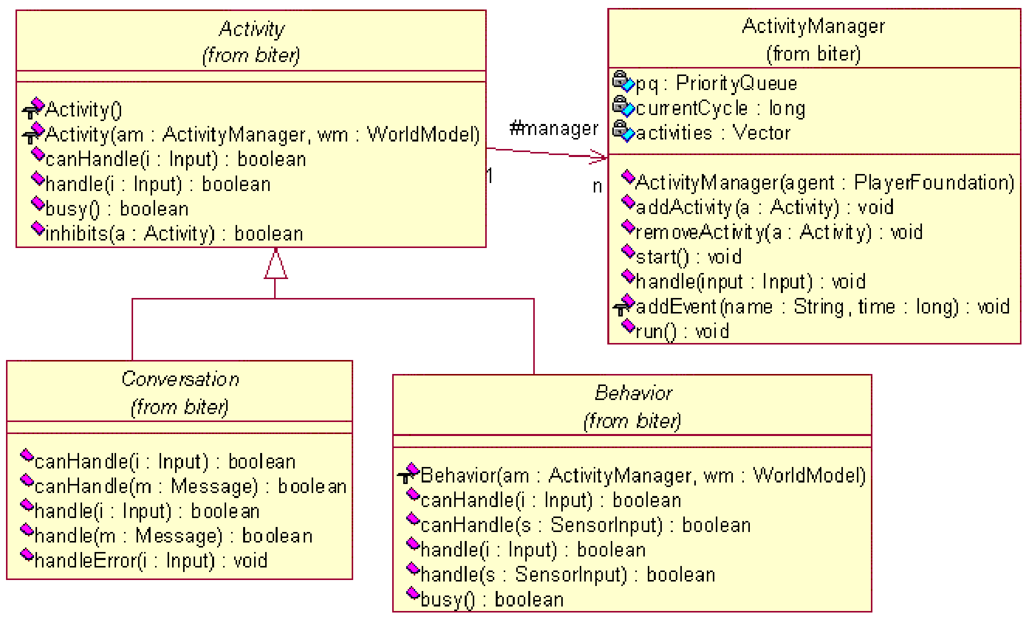
Activity
(from biter)
Activity()
Activity(am : ActivityManager, wm : WorldModel)
canHandle(i : Input) : boolean
handle(i : Input) : boolean
busy() : boolean
inhibits(a : Activity) : boolean
#manager
1
n
ActivityManager
(from biter)
pq : PriorityQueue
currentCycle : long
activities : Vector
ActivityManager(agent : PlayerFoundation)
addActivity(a : Activity) : void
removeActivity(a : Activity) : void
start() : void
handle(input : Input) : void
addEvent(name : String, time : long) : void
run() : void
Conversation
(from biter)
canHandle(i : Input) : boolean
canHandle(m : Message) : boolean
handle(i : Input) : boolean
handle(m : Message) : boolean
handleError(i : Input) : void
Behavior
(from biter)
Behavior(am : ActivityManager, wm : WorldModel)
canHandle(i : Input) : boolean
canHandle(s : SensorInput) : boolean
handle(i : Input) : boolean
handle(s : SensorInput) : boolean
busy() : boolean

Figure 2: UML class diagram of the activities and activity manager.

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.

Allowing all activities to create new activities can lead to some unfortunate situations. For example, an agent might generate two contract-net conversations which end up compromising the agent to do two tasks at the same time. This type of problem arises because the various activities are assuming they have a resource, such as the agent's time, available to them. The standard solution to this problem is for the activities to reserve or lock the resource, so that others cannot use it.

Another way to control the number and type of conversations that are active at any time is by using a conversation factory [13, Factory Method]. With this method a conversation asks the factory to create a new conversation, giving it the needed parameters such as role, initial message, etc. The factory can then decide what type of conversation to create and whether or not the proposed new conversation will interfere with an ongoing conversation. If there is a problem, the factory can refuse to create a new instance.

The agent conversations also need to be fault tolerant. They need to function well in the presence of errors. These errors can take many forms: a message lost by the transmission medium, an agent that stops functioning, a malicious or malfunctioning agent that sends too many messages or fails to reply, a message that takes too long to arrive, etc. In an agent that implements many conversations, we do not want to have to implement the error handling logic in each one of them. One proposed solution to this problem is the creation of exception handling agents that monitor the network and the agents' conversations [9]. Our solution is to provide all agents with a simple exception handling code that all conversations can use.

Specifically, all the errors mentioned above manifest themselves as either a failure to receive a message or the receipt of too many messages. Under the GAA, whenever a conversation sends a message for which it expects a reply from another agent it also creates a special event which it adds to the priority queue. When the reply arrives the agent removes the event. However, if the reply takes longer than the time specified in the event then the conversation will receive the event before any reply. The receipt of this event indicates to the conversation that it has timed-out on waiting for the reply.

The received timeout event signifies that there could be any number of problems: the reply message could have been lost in transit, the agent could have died, or the agent could be refusing to answer. Depending on the particular system, it might be possible to differentiate between these situations. For example, in a networked environment we could keep track of the network load and deduce the probability that the packet was lost, or in a closed system we could inquire as to whether or not a particular agent is still active or has been certified as a member of the system.

Of course, we do not want each conversation to re-implement such error handling and verification code. Instead we can define a `handleError` function in the Conversation class and place the error handling code there. Whenever a conversation times out on a reply it simply calls this function which will instruct it on what to do next, such as re-sending the message or aborting the conversation.

The arrival of unexpected messages can also be consid-

```
$pq$ = new PriorityQueue of Event
**loop**
  **while** (next = $pq$.top().time) $\leq$ current time **do**
    handle(next)
  **end while**
  input = get data from sensors
  handle(input)
**end loop**
```

Figure 3: ActivityManger.run()

ered an error. For example, a rogue agent might be sending a lot of messages in the hopes of overwhelming other agent, a situation specifically relevant to open systems. Therefore, we suggest that the creation of new conversations be monitored with calls to the error handling function. That is, if the activities spawn new conversations by themselves, they should only do so if a call to `handleError` determines that this is safe. On the other hand, if a factory produces the new conversations then it is the factory's responsibility to call the error handling function. In either case, the error handling function can keep a list of the agents that have requested new conversations and block any agents that seem to be abusing the system.

2.3 Activity Manager

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 `run` method of the manager in shown in Figure 3.

Most of the work, however, is done by the `handle` function, shown in Figure 4, which determine which of the activities will actually handle the input. The algorithm it implements echoes the type of control mechanism implemented by subsumption and BDI 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. Multi-step activities must make sure they implement their `canHandle` functions so as to enable them to be selected on successive time steps.

Each agent that implements a GAA needs to instantiate one copy 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. A user of the GAA architecture should not need to modify the manager. All the control knowledge is stored in the `canHandle` and `inhibits` methods of the activities the user must define.

```
$input$ = 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**
$chosen$ = pick randomly from $uninhibited$
**if** $chosen$.handle() **then**
  removeActivity($chosen$)
**end if**
```

Figure 4: ActivityManager.handle(Input input)

3. THE GAA AS A SUBSUMPTION ARCHITECTURE

The GAA, while object-oriented and modular, can be made to function as a pure subsumption architecture [7] without any changes to the design, only requiring the user to be careful when defining activities. The GAA already implements a subsumption mechanism via the use of the `inhibits` function, so we only need to make sure that the agent does not maintain a state. Since the architecture is not responsible for the implementation of the actual activities, it is the user's responsibility to make sure that none of the activities maintains any state variables that keep their values across successive invocations. Also, the `inhibits` function should make its decisions based solely on the type of other activity. In this way, we arrive at a function that is equivalent to the one use in subsumption architectures.

4. THE GAA AS A BDI ARCHITECTURE

The GAA can also function as a BDI architecture, once some simple extensions are made to the basic architecture. Specifically, a *desires* container class is needed to hold the set of active desires for the agent. This class should implement access functions to carry out basic tasks such as adding a new desire, removing a desire, and determining if a given desire is active. The agent should have a reference to this class.

Once a desire container has been implemented, the user of the GAA must make sure that the `canHandle` method of each activity returns true only if the goal that the activity achieves is part of the set of active desires. Also, once the activity achieves the desire, it is responsible for removing it before returning. That is, each activity is associated with a desire, and the activity itself is responsible for firing only when that desire is active and for removing the desire once it has been achieved.

The GAA architecture does not impose any intention semantics. There exist many BDI logics with various intention semantics, and there exist many BDI-based systems each of which implements different methods for determining when an intention should be dropped and when a new intention should take priority. The GAA does not claim to implement any such method. It is up to the user to decide which method is best for his application and to implement it. For example, a user might decide to associate a priority number with each desire/goal and then modify the `inhibits` methods of his activities so that activities inhibit each other based on the priority of the goal they achieve. For example, an activity can inhibit all other activities that achieve goals of lower priority.

5. BITER: AN IMPLEMENTATION OF A GAA FOR ROBOCUP

The goal of the RoboCup initiative is to spur fundamental research in the areas of AI and intelligent Robotics, by providing a standard problem of both sufficient complexity, and near universal familiarity. This goal is met with the game of soccer, which is played worldwide. The success of RoboCup in achieving its goal can be inferred by the growing participation at the RoboCup world championship competition. These RoboCup competitions are partitioned into several leagues. These leagues include legged, medium and small robots, and one for software simulation. The Biter framework was designed as a GAA implementation of a soccer player for the simulation league, to be used as part of a graduate University course in multiagent systems.

The RoboCup simulation league allows teams, composed of software agents, to compete in head-to-head competition. The architecture of the simulation league consists of a centralized server, a monitor application, and individual agents, which are the players of the game. The server manages game play and provides sensory input to the players via a messaging service built on UDP communications. This sensory input consists of messages that contain visual information, auditory information, and information from the referee. In response to this sensory stimulus from the simulated field of play, a player takes action through a message-based set of effectors that notify the server of the player's activity. A player can indicate that it is kicking the ball, dashing on the field, and shouting information to other players. A detailed description of the RoboCup simulation league architecture and messaging protocol can be found in the Soccerserver Manual [11].

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 RoboCup within the classroom setting. Since Biter is an instantiation of the GAA, it allows students to explore both strong and weak notions of agency[26]. Biter also provides a number of low-level ball handling skills as well as higher-level skill based behaviors. Additionally, many functional utility methods are provided which allow students to focus more directly on planning activities. Biter is written in Java 2. The primary features of the language leveraged by the Biter code are a native support for multiple threads, a built-in network communications capability, and a 2-dimensional graphics library.

5.1 Biter's World Model

Any implementation of the GAA will need a way to represent the objects in that agent's world. Since the type of objects vary greatly from domain to domain, the GAA cannot specify how they should be represented. In the RoboCup domain, for example, it has become clear that agents need to build a world model [23].

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. The following sections will further describe the implementation of Biter's world model.

5.1.1 World Model Update

As previously mentioned, Biter was designed to provide a framework for student exploration with the theory and techniques of multiagent systems. Due to this consideration, Biter's world model needs to support the both state and stateless agent architectures. At first blush, it seemed difficult to resolve these conflicting goals within a unified framework; however, when one realizes that a stateless agent simply implies that it maintains no state history, a straightforward implementation becomes obvious. As sensory information, about dynamic objects, is placed into Biter's world model it is time stamped. Biter implements a command-line argument which indicates the maximum age of world model data. If world model information is not allowed to age, it is discarded from the player's memory during each update cycle. Additionally, this timestamp allows for the easy detection of stale data within the world model. Stale data is defined to be a world model element that was reported at some time in the past, but has not been updated in subsequent cycles. When stale world model elements surpass the age threshold they are removed. By allowing the specification of this threshold at run-time, students can experiment with this parameter and see how it impacts the behavior of their player.

5.1.2 World Model Access

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 [13] 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 [3, pp. 162–164]. 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. For example, the algorithm for computing the player's absolute field position requires access to the flag and goal objects in the sensory input. Specifying the filtering criteria as ([fF]lag—[gG]oal).+ creates an iterator that only returns the desired elements. 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.

5.1.3 World Model Display

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 runtime visual display of a player's internal view of his environment. The Java language, having built in windowing routines and 2-dimensional graphics primitives, simplified the development of this display capability. 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 students countless hours of fruitless troubleshooting and helps them focus on other multiagent system implementation issues.

5.2 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 back-seat 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. The software, as well as full UML diagrams and a Javadoc API, is available for download [1]. We are interested in receiving feedback from others who choose to use Biter in an academic setting.

6. RELATED WORK

While there are many agent architectures in the literature, most of them implement a specific agent system. The user of these systems is expected to download the provided software and build his agents as extensions to the system. Sometimes these architectures take the form of software libraries that the user can link to, such as JATLite[16] and JAFMAS [8], other times they provide graphical user interfaces which the user is expected to use for building his agents, such as ZEUS [19]. As such, these agent architectures provide a finished product which commits the user to a specific structure, a specific language, and a specific set of limitations. Our GAA, in contrast, is meant as a general design to be implemented by the user using any object-oriented language he desires. Of course, our GAA borrows some of the best ideas from all the existing agent systems and tries to present them to the user in their simplest form.

The JADE agent framework [4], for example, implements a FIPA-compliant agent system. Like the FIPA standards [20], JADE focuses on providing inter-agent communication services such as the Agent Management System and Directory Facilitator required by FIPA, as well as the code necessary for interfacing with these services. Such communication services are orthogonal to the functionality we provide with the GAA. That is, it would be feasible to use them with a GAA agent. However, JADE goes further and provides a scheduler, akin to our ActivityManager, which schedules behaviors. The JADE scheduler, however, only "carries out a round-robin non-preemptive policy among all behaviors available in the ready queue." That is, it does not allow for behaviors to inhibit each other and does not provide support for more complex dynamic behavior selection. The JADE system also does not provide explicit support for conversations.

These systems present just a sampling of the available agent architectures, other systems include the Cougaar Cognitive Agent Architecture [2], the Open Agent Architecture [18], the ZEUS agent building toolkit [19], and the MadKit platform [15], among many others. There are also many commercial systems now available for the implementation of agent systems, as analyzed in [22]. All these systems provide complete solutions which, by necessity, commit the user to a particular way of implementing communications or, in some cases, of implementing the internal control flow of the agents. They are useful for users that desire a quick solution and whose needs closely match the features by the software. The GAA, on the other hand, should be used by those who wish to implement a complete agent, or who find that none of the existing software solutions are satisfactory, or who are not sure what type of control mechanism will be best suited to the domain and want a flexible architecture that will accommodate future changes in requirements.

7. SUMMARY

We have introduced our Generic Agent Architecture along with Biter—an implementation of the GAA for the RoboCup domain. The GAA incorporates an elegant object-oriented design meant to handle the type of interactions that an agent in a multiagent system can expect. The type of interactions include reactive responses, long-term behaviors, and conversations with other agents. We have also shown how the GAA is generic enough so small modifications in its implementation can lead to a purely reactive agent or to a BDI agent. The GAA is meant to be used by designers and researchers who want to implement a complete agent and want the flexibility of changing control semantics as the project progresses while still maintaining a clean separation between behavior and control knowledge. It is also useful as a didactic tool for teaching multiagent systems design. Finally, our Biter implementation serves as a proof of concept and illustrates the added functionality that a user of the GAA must implement for a specific domain.

8. REFERENCES

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

[2] The cougaar agent architecture. `http://www.cougaar.org`.

[3] D. Bailey. *Java Structures, Data Structures in Java for the Principled Programmer.* McGraw-Hill, 1999.

[4] F. Bellifemine, A. Poggi, and G. Rimassa. Developing multi-agent systems with JADE. In *Proceedings of the Seventh International Workshop on Agent Theories, Architectures, and Languages*, 2000.

[5] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide.* Addison-Wesley, 1999.

[6] M. E. Bratman, D. J. Israel, and M. E. Pollack. Plans and resource-bounded practical reasoning. *Computational Intelligence*, 4:349–355, 1988.

[7] R. A. Brooks. Intelligence without representation. *Artificial Intelligence*, 47:139–159, 1991.

[8] D. Chauhan. *JAFMAS: A Java-based Agent Framework for Multiagent Systems Development and Implementation.* PhD thesis, University of Cincinnati, 1997.

[9] C. Dellarocas and M. Klein. An experimental evaluation of domain-independent fault handling services in open multi-agent systems. In *Proceeding of the Fourth International Conference on MultiAgent Systems*, pages 95–102, 2000.

[10] M. d'Inverno, D. Kinny, M. Luck, and M. Wooldridge. A formal specification of dMARS. In Singh, Rao, and Wooldridge, editors, *Intelligent Agents IV: Proceedings of the Fourth International Workshop on Agent Theories, Architectures and Languages*, Lecture Notes in Artificial Intelligence, 1365, pages 155–176. Springer-Verlag, 1998.

[11] E.Corten, K. Dorer, F. Heintz, K. Kostiadis, J. Kummeneje, H. Myritz, I. Noda, J. Riekki, P. Riley, P. Stone, and T. Yeap. *Soccerserver Manual*, 1999.

[12] M. Fisher. A survey of concurrent METATEM: the language and its applications. In *Proceeding of the First International Conference on Temporal Logic*, pages 480–505. Springer, 1994.

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

[14] M. P. Georgeff and F. F. Ingrand. Monitoring and control of spacecraft systems using procedural reasoning. Technical Report 03, Australian Artificial Intelligence Institute, Melbourne, Australia, 1989.

[15] O. Gutknecht, F. Michel, and J. Ferber. MadKit: A generic multi-agent platform. In *Proceedings of the Fourth International Conference on Autonomous Agents*, pages 78–79, 2000.

[16] H. Jeon, C. Petrie, and M. R. Cutkosky. JATLite: A java agent infrastructure with message routing. *IEEE Internet Computing*, Mar/Apr 2000.

[17] J. Lee, M. J. Huber, P. G. Kenny, and E. H. Durfee. UM-PRS: An implementation of the procedural reasoning system for multirobot applications. In *Proceedins of the Conference on Intelligent Robotics in Field, Factory, Service, and Space (CIRFFSS)*, pages 842–849, Houston, Texas, March 1994.

[18] D. L. Martin, A. J. Cheyer, and D. B. Moran. The open agent architecture: A framework for building distributed software systems. *Applied Artificial Intelligence*, 13(1-2):91–128, January-March 1999.

[19] H. Nwana, D. Ndumu, L. Lee, and J. Collis. ZEUS: A tool-kit for building distributed multi-agent systems. *Applied Artifical Intelligence Journal*, 13(1):129–186, 1999.

[20] P. D. O'Brien and R. C. Nicol. FIPA: towards a standard for software agents. *BT Technology Journal*, 16(3):51–59, 1998.

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

[22] P.-M. Ricordel and Y. Demazeau. From analysis to deployment: a multi-agent platform survey. In *Proceedings of the Workshop on Engineering Societies in the Agents' World.* Springer-Verlag, 2000.

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

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

[25] G. Weiß, editor. *Multiagent Systems : A Modern Approach to Distributed Artificial Intelligence.* The MIT Press, 1999.

[26] M. Wooldridge and N. R. Jennings. Intelligent agents: Theory and practice. *Knowledge Engineering Review*, 10(2), 1995.