A Reusable Component Architecture for Agent Construction *

Bryan Horling Victor Lesser

UMass Computer Science Technical Report 98-30

May, 1998

Abstract

A generic, component based architecture is proposed as a basis for designing the agents used within Multi-Agent Systems. The architecture, based on Sun's Java Beans, is explored, and both domain independent and applied component examples are described in detail. Designs for theoretical new components for the applied agent are also proposed and examined.

1 Overview

Component based architectures are a relatively new introduction to software development. They attempt to effectively encapsulate the functionality of an object while respecting interface conventions, the goal being to easily combine groups of components to create stand alone applications. This type of design promotes software reusability - the ability to painlessly transport source code from one project to another - which is a long sought after but infrequently achieved goal of software engineering. In this paper I will describe such a component architecture designed for use in the domain of agent construction.

Much of the research which goes into Multi-Agent Systems deals with their behavior and organization, often viewing the implementation of these concepts as secondary. While the intellectual contributions of such research are clearly the more important facet, I believe this practice has several weaknesses, both with problems stemming from *ad hoc* agent construction, and in the act of construction itself.

A recently emerging and exciting trend in the field of computing has been the combination and synthesis of existing technologies to solve difficult problems [7, 11]. On paper, this may not appear to be particularly hard, but in practice it may result in months of reimplementation because of incompatibilities between existing systems. If the original agent systems, and their respective technologies, had been originally developed using a clean modular style, this problem could have been avoided. Such a programming style is also beneficial to everyday architectures, as it both

*This material is based upon work supported by the National Science Foundation under Grant No. IRI-9523419. Disclaimer: Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

improves maintainability and facilitates changes. Too often, though, final systems are built using prototype code which has outlived its original purpose.

There exists a common pool of functionality that many agents possess, such as the ability to communicate across a network, locally store and access information, and initialize their local state. Though individually trivial, the sum of these functions can represent a sizable body of work which is usually rewritten every time an agent is created. Among subgroups of agents, even more functional overlap may arise because of their common use of an operating environment and data representation and manipulation. Rather than regenerating this support code, it seems clear that some mechanism of reusing old code would be very useful, if it could be made appropriately domain independent.

Code reuse and modularity are not new ideas in the field of computer science [1]. The mechanism which I have implemented, though, is a level of granularity above this. Instead of working with simple objects, agents using my framework are built up from reusable *components*. A component is differentiated from an object in that they are typically more robust and self-contained, and adhere to particular naming and behavior conventions to facilitate interoperability. Component technology is particularly applicable to experimental multi-agent systems because changes between agent variants can usually be isolated to certain functional areas; replaceable modular components can therefore provide an almost plug-and-play form of testbed.

The framework which I have developed is written in Java [6], and builds upon Sun's Java Beans specification [4, 5, 3] by strengthening inter-component relationships and adding more control mechanisms. The Java Beans architecture was chosen as a starting point because of its clean yet powerful organization and integration mechanisms. Agent development and code reuse will also be facilitated by the availability of visual building tools originally developed for Java Bean compatible components.

The following sections should give a thorough overview of why the framework was created and how it can be used. I will start out by describing the motivation for this project. The architecture section describes how the framework is designed and operates. Following this, each of the components in an existing agent will be discussed in detail, and hypothetical new components will be described. Future improvements and additions will be covered in the conclusion.

2 Motivation

The motivation for the agent framework began with the creation of the Multi-Agent Survivability Simulator (Mass) [9]. Mass is a flexible execution environment which we designed to simulate the possible faulty or hostile conditions under which an agent might function. By accurately simulating these conditions, we can then create and test effective algorithms to deal with adverse situations, the end result of which is to make our multi-agent systems more robust.

The simulator consists of a centralized controller object, to which a number of possibly remote agents connect. The controller is then responsible for simulating or providing those aspects of the environment which the agent must interact with. These controlled aspects include such things as method execution, agent knowledge, message transfer and environmental constraints. Therefore, some aspects of the agent, such as communication and execution, must necessarily know of the existence of the simulator. Other parts, such as problem detection and diagnosis, will be "live" and unaware of the simulator, to more accurately test their behavior.

I developed an initial *ad hoc* agent to satisfy these conditions. It worked as expected within

the Mass environment, but it became clear that the design was not generic enough to support the different kinds of agents which we planned to develop. The monolithic design of the initial agent did not promote simple modification, and significant extensions to any one agent would quickly make the internal architecture incompatible with other available agents. Since it was our hope to be able to make functional changes, such as using new coordination or diagnosis strategies, *en mass* to the agents running in the system, it was decided that the initial agent design would not be sufficient. Our efforts creating the simulation environment and initial agent design are documented in the appendix *Survivability Simulator for Multi-Agent Adaptive Coordination*.

Thus, a new design was needed for the agents working within the Mass environment. It needed to be flexible and extensible, and yet maintain separation between mutually dependent functional areas to the extent that one could be replaced without modifying the other. It also needed to be easily customizable, allowing a wide range of agent functionality to be implemented with simple confined alterations. The component based architecture described in the following sections was created to satisfy these requirements.

3 Architecture

The agent framework is based on Java Beans, Sun Microsystem's component model architecture. For purposes of clarity, when discussing aspects of the agent framework derived directly from Sun's architecture, components will be referred to as beans. The Java Beans framework consists of a set of classes and an API specification for the beans themselves. The classes in the *java.beans* hierarchy exist primarily as support for bean functionality, providing bean description interfaces, basic event classes and other utilities. For the most part, usage of these classes is not necessary within the agent framework, although more visually advanced components may use them.

3.1 Java Beans API

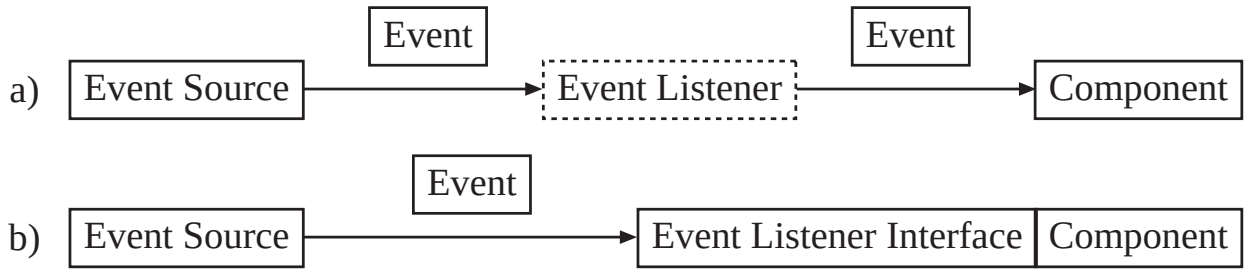

Figure 1: Event propagation model for (a) "dynamic" links created with beanbox and (b) "static" links specified in the component source code.

More germane to the agent framework is the Java Bean API, which provide behavior and naming specifications that beans must adhere to. Essentially, the API is a set of naming conventions which allow both application construction tools and other beans to easily manipulate the bean's state and make use of its functionality.

Most beans will have a set of paired state accessor functions, which are used to both retrieve and modify data within the bean. For each publicly modifiable variable, there should exist a pair

of functions, one for setting the contents of the variable and another to retrieve it. The API specifies that these functions should be named *setVariableName* and *getVariableName*, respectively, where VariableName should be replaced with some concise description of the targeted variable (e.g. *set/getRemoteHostName*). Adherence to this convention not only enforces encapsulation, but also allows automated tools to easily recognize the configurable parts of a given bean. Application construction tools can then use this knowledge to provide intuitive graphical interfaces to simple data types without the need for explicit supporting code.

Another important feature of the Java Beans architecture is its notion of event streams. Each bean may support one or more event streams, to which other beans may subscribe. The source bean then generates events at runtime, indicating for instance that an action has taken place or a state change has occurred, and propagates them to only those other beans which have registered as being interested in those event types. Such an event generating bean is said to *fire* events to a number of other *listener* beans. This feature is important because it allows loose connections to form dynamically between beans. These loose connections in turn facilitate the construction of functional applications from a number of smaller modules, a key goal in the agent framework. A harder connection, such as a direct method invocation, would not be as flexible as this arrangement, and would likely require additional code changes when integrating the components. Similar to the state accessor methods, the event stream interface also makes use of naming and construction conventions. The source bean must contain two methods for managing its listener set: *addEventListener* and *removeEventListener*, where EventListener is replaced with the type of event listener (e.g. *add/removeActionEventListener*). Each of these methods takes a single listener object as an argument; the source bean is responsible for managing and making use of the list of listeners. On the other end of the connection is the listener itself, typically represented by an abstract interface (e.g. *ActionEventListener*), which a given bean can implement. Another class specifies the event itself (e.g. *ActionEvent*), which can contain whatever methods or state necessary to convey the meaning of the event. At runtime, event generation therefore starts with the source producing and firing an event object to each of its listeners. The firing mechanism itself calls a specific method in each of the listeners, which can ignore or act upon the event as necessary.

It is important to note that if one is using a bean-aware construction tool, such as Sun's *beanbox*, it is not necessary for beans to directly implement the listener interface for an event type to make use of a particular event stream. Instead, these tools construct adaptor classes on the fly which implement the desired interface, and in turn call an arbitrary method within the intended recipient when an event is received. This design can be seen in Figure 1 (a), with the dashed component represents the beanbox constructed adaptor. These adaptors are transparent to the user, and are incorporated into the final Java applet automatically[1]. Using a graphical tool to make these loose connections can be quite useful, but during development it can be arduous to have to re-specify all of these links each time the agent is to be compiled. To overcome this problem, I have added support within the agent framework to generate code specified, inter-component connections at runtime. To see how this works, the construction and initialization process of an arbitrary component will explored in the next section.

[1] At this time, agents constructed in the framework using the beanbox tool are generated as applets, a restriction which will be addressed in a later section.

3.2 Dependency Registration and Checking

To ensure compatibility with the agent framework, components should extend *agent.base.AgentComponent*, a base class which provides the standard component interface and utility functions. When each component is constructed at runtime, the default constructor method located in the AgentComponent class will also be called, which in turn sets a component descriptor array. The descriptor array, which defaults to the classname of the component, contains one or more strings indicating the functionality of the component. As an example, a network-aware messaging class could be described with "Communicate", whereas a generic logging component could be called "Log". Once the descriptor is set, the component registers itself with the State component, a mandatory object which will be discussed more thoroughly in a later section. This State component then serves as a utility for creating inter-component connections. A problem solving class can, for instance, use State to obtain a handle to the Communicate component which was registered earlier. With this handle it can then add itself to Communicate's event streams (to watch for message deliveries and arrivals) or use it to invoke any of Communicate's public methods (e.g. *sendMessage*). When searching for other components, substring descriptor matches may also be used, which permits even more flexibility if clever descriptor strings are used. Consider the case where all of the installed components make use of a logging component whose descriptor is "Log". State would normally match this to *agent.simplest.Log*, but will also match it to *agent.simplest.FileLog* or *agent.advanced.NetworkLog*. This means that even though "static" descriptions of target components are specified in the source code, the reference may actually map to a different component, depending on what is available. If consistent interfaces are used for related components, this allows the designer to simply replace one module with another to effect a different behavior.

Given that components are now allowed to somewhat statically define a connection to a component which may or may not exist at runtime, dependencies can exist in the agent which are not checked by the compiler. One solution to this problem would be to accept the fact that unresolved dependencies will throw some sort of exception sooner or later during execution. To handle this situation in a more elegant manner, I have formally introduced the notion of component dependencies. Each component is expected to announce its dependencies in its constructor method, using the *addDependency* method provided in the base AgentComponent class. Once all the components have been constructed, the Control component, another mandatory object, verifies each of these dependencies before beginning execution. This ensures that missing dependency targets will be discovered at program commencement and handled cleanly.

3.3 Initialization and Execution

Two more phases follow the registration and dependency checks: component initialization and execution. During the initialization phase, the *init* method of each registered component is called, in no particular order. The only guarantee made about the system state at this stage is that all other components have registered successfully. Because of this, initialization typically will consist of setting up the internal state for each component and forming event stream connections to other components. Unless explicitly permitted, components should refrain from calling other components' methods during this phase, as their potentially unstable internal state may lead to race conditions.

During the final execution stage, the *begin* method is called once for each of the components. Any continuous or periodic actions should be run here. Examples of this include a communications

module monitoring for incoming messages or connections, or a problem solving module traversing its search space. Activity spawned within the begin methods provides the driving force for the entire system, as they not only are a direct source of activity, but also indirectly affect the activity of other modules through generated events.

3.4 Control

The Control component, one of two "mandatory" components which must be included for a constructed agent to function correctly, is responsible for regulating the activity within the agent. As mentioned above, Control begins its work after the Java applet has constructed each component, by checking the dependencies which the components (including itself) have registered. If the dependency check succeeds, Control then initializes each component in no particular order, after which it instructs them to start execution.

The Control module also serves several other functions. The first is to provide a mechanism for running the agent outside of the normal applet/appletviewer environment. To do this, Control has a normal *main* method which, when called, instantiates the Java applet internally without the use of an external viewing program. This permits the agent to be directly run as an application, reducing start up time and bypassing security restrictions. Control also serves as a centralized thread registry. As new threads are created by individual components, they are expected to register them with the Control component. This registry then provides a way of controlling just the threads related to agent execution[2], which includes suspending, resuming, and killing them. Control also provides methods for reseting, restarting and quitting the agent.

3.5 State

The second mandatory component is State, which, in addition to the registry of installed components, keeps track of the properties the components make use of. State functions as a global hashtable, to which components can add arbitrary data or retrieve previously stored information. State is also responsible for firing events when properties are added, changed or removed, which allows components to base behaviors on certain state changes. Any piece of information which might be of use to other components should be stored in State's property table.

The property table also serves as the repository for runtime configuration information, which the State component gathers during its initialization. Components are able to individually specify what parameters they are interested in with the static method *addParameterInfo*, which allows the State component to remain component-independent. State then uses this list of recognized parameters to search for keywords either in its HTML context (if applicable) or in the system property list. Since Java's System property list incorporates variables set on the command line, this mechanism provides an easy way to change the runtime behavior of previously compiled components.

3.6 Implementation

The general design described above has been implemented and tested. To date, the entire framework consists of 32 classes which contain more than 300 methods in 4500 lines of code. The basic building

[2]I assume that if the users wants to control *all* the available threads that they can use the normal *java.lang.Thread* methods

components of the framework comprise two Java packages: *agent.base* and *agent.simplest*. The five agent.base classes define the intrinsic properties the individual parts of the framework must have, acting as the base classes to be extended in an actual agent. These classes serve to both enforce naming conventions and provide default functional content for methods the designer may choose to not override.

The 16 classes in the agent.simplest package provides all the functionality to make a basic agent. It includes services for communication, execution, and logging and implements the mandatory state and control components. The simple components should be viewed as a robust starting point for an actual agent, providing direction for the designer of a more customized agent. Much of the functionality required by an agent exists here, making customization easy if the designer adopts this particular organization. As we will see in the following section, only minor additions were necessary to these classes to create a basic agent compatible with our Mass simulation environment. Similarly, it is expected that only minor modifications would be necessary to create a simple independent agent, using this starting point.

4 An Example Agent

In this section I will explore the construction of a generic agent using the framework described above. Each component will be described, including a brief synopsis of its characteristics, along with a more formal description of its behavior and capabilities. Unless otherwise noted, subclasses should be assumed to inherit all of the traits of their parents. In the synopsis listings, only derivations from the parent class will be noted[3].

The descriptions below include components for both the simple base agent described above and a generic Mass-compatible agent. The Mass agent represents a simple example of how the general framework and simplest components can be adapted to specific working conditions. If possible, it may be illustrative to use the actual Mass agent source code as a reference while reading the following section.

4.1 Communicate

Name:	agent.simplest.Communicate
Extends:	agent.base.AgentComponent
Depends:	State, Log, Control
Parameters:	Host, Port
Fires:	MessageEvent
Listens:	State:PropertyChangeEvent

The Communicate component is responsible for handling the network message activity within the agent. When the component is started, it either opens up a connection to a remote host, or listens on a numbered port. After this, it spawns and registers a new thread, which will monitor the newly opened connection. As messages arrive, they are decoded and placed in a Message object, which is in turn stored inside a MessageEvent object suitable for distribution to any listeners which

[3]In particular, a child who's *Fires* description is None will still fire all of the same events as its parent.

might be present. Communicate also supplies a *sendMessage* method, which other components may use to send messages.

The PropertyChange event stream is used to watch for changes to the two parameter variables the component is interested in.

Name:	agent.mass.Communicate
Extends:	agent.simplest.Communicate
Depends:	None
Parameters:	None
Fires:	None
Listens:	None

The Mass Communication component differs because of its support for the simulation environment. The component has modified the way messages are sent and retrieved, reflecting the fact that KQML is used to encode interactions with the simulator. Message pre and post-processor methods were also added to watch for simulation-specific messages who's content must be handled at certain times during the clock cycle.

Message receipt was further modified to reflect the fact that activity should only take place while the agent is being pulsed. To do this, MessageEvent firing for received messages are queued until a pulse is received, when the queue of MessageEvents is flushed to the other components. In this sense, only messages received before the pulse is generated are considered to be active during that time period. Control messages, the non-scenario messages sent by the simulator (e.g. reset, disconnect), are always fired immediately upon receipt.

4.2 Control

Name:	agent.simplest.Control
Extends:	agent.base.AgentComponent
Depends:	State, Log
Parameters:	None
Fires:	None
Listens:	None

This Control component is identical to the one described in the architecture section.

Name:	agent.mass.Control
Extends:	agent.simplest.Control
Depends:	Communicate
Parameters:	None
Fires:	None
Listens:	Communicate:MessageEvent

The Mass Control component was modified to enable it to listen to Communicate's Message event stream. It watches this stream for both reset and disconnect messages arriving from the centralized controller.

4.3 Execute

Name: agent.simplest.Execute
Extends: agent.base.AgentComponent
Depends: Log
Parameters: None
Fires: ActionEvent
Listens: None

The Execute component is responsible for actually performing the actions associated with the agent, which could range from direct responses to queries to simply performing tasks laid out in a global schedule. The simplest Execute component does not do much of anything, as the specific mechanism for executing tasks is usually domain dependent. Instead, it functions more as a template, so that component writers will have an idea what to expect in derived Execute components.

Name: agent.mass.Execute
Extends: agent.simplest.Execute
Depends: Communicate, State
Parameters: Location, Display, RunDisplay
Fires: None
Listens: Communicate:MessageEvent, State:PropertyChangeEvent

The Mass Execute component builds on the simplest template described above by adding the functionality needed to simulate execution. It uses a PropertyChange event stream from State to watch for changes in the global TÆMS [2] task structure, which may or may not contain an execution schedule. If a schedule is found, it selects the first method from the list and sends it to the simulation controller to be executed. The Execute component then makes use of a Message event stream from Communicate to watch for the completion of its task, which the controller will send to the agent at the correct time. Upon receipt of this completion notification, Execute will read in the method's execution characteristics and fire an ActionEvent. Because this behavior is encapsulated within the Execute component, the fact that the methods are simulated is transparent, which makes the overall design less domain dependent.

As noted in the synopsis, Execute also recognizes three new parameters, which it uses to produce a visual display of the agent at the simulator. It continues to monitor changes to these variables using the same PropertyChange event stream mentioned above, notifying the simulator if an update occurs.

4.4 Log

Name: agent.simplest.Log
Extends: agent.base.AgentComponent
Depends: State
Parameters: LogLevel
Fires: None
Listens: State:PropertyChangeEvent

The Log component is a simple logging class, which provides generic methods to facilitate both general and event stream logging to the standard error stream. Each string to be logged has associated with it a level, assigned by the caller. If this level is less than or equal to Log's internal log level, the string is recorded, otherwise it is thrown out. Log messages are automatically tagged with the actual time, execution thread name, agent name and agent time. The latter two are obtained by watching for changes to the Time and Name properties in the PropertyChange event stream.

Name:	agent.simplest.FileLog
Extends:	agent.simplest.Log
Depends:	None
Parameters:	None
Fires:	None
Listens:	None

FileLog is a trivial subclass of Log, with the exception that a file name is specified as the destination for the log messages, rather than standard error. Both overwrite and append modes are supported. Note that a component dependent on "Log" could use either FileLog or its base component, Log, without changing a single line of code. This is good example of how substring name matching and a common interface can be exploited to permit seamless component replacement.

4.5 State

Name:	agent.simplest.State
Extends:	agent.base.AgentComponent
Depends:	Log
Parameters:	Name
Fires:	PropertyChangeEvent
Listens:	None

The State component is identical to that described in the architecture section.

Name:	agent.mass.State
Extends:	agent.simplest.State
Depends:	Communicate
Parameters:	Name
Fires:	None
Listens:	Communicate:MessageEvent

The Mass State component adds support for remote property queries and changes by monitoring Communicate's Message event stream. It also allows the system designer to specify an arbitrary number of constraints, represented with a supplied Constraint class, both within a configuration file or as supplied remotely by the simulation controller.

4.6 Problem Solver

Name: agent.mass.ProblemSolver
Extends: agent.base.AgentComponent
Depends: Log, State, Communicate, Execute
Parameters: Name
Fires: PropertyChangeEvent
Listens: Communicate:MessageEvent, Execute:ActionEvent

In a typical agent, the problem solver will act as an instigator to other components. It may be responsible for discovering and evaluating problems or goals, and determining how best they can be solved or achieved. Within the Mass architecture, though, the problem solving component has been replaced with a more domain independent mechanism relying on pre-formed task structures rather than complex behavioral algorithms. The job of creating task structures for individual agents has been delegated to a task generator, housed within the centralized simulation controller. When a simulation scenario begins, this generator is responsible for distributing the task structures, which the agents then make use of as if they were created locally.

The Mass problem solving component, then, is responsible for receiving this task structure from the generator. Once the structure has been obtained, it is run through a design-to-criteria scheduling utility [10], which attempts to produce a schedule satisfying the agent's execution criteria. The schedule is then incorporated into the task structure by the problem solver, and placed in State's property table.

It should be noted that with the simple replacement of this component, the agent could exhibit very different behavior, yet still function correctly within the Mass environment. By separating the actual communication and execution processes, a substantial problem solver with interesting behavior could replace this component without the need to modify other parts of the agent.

4.7 Integration

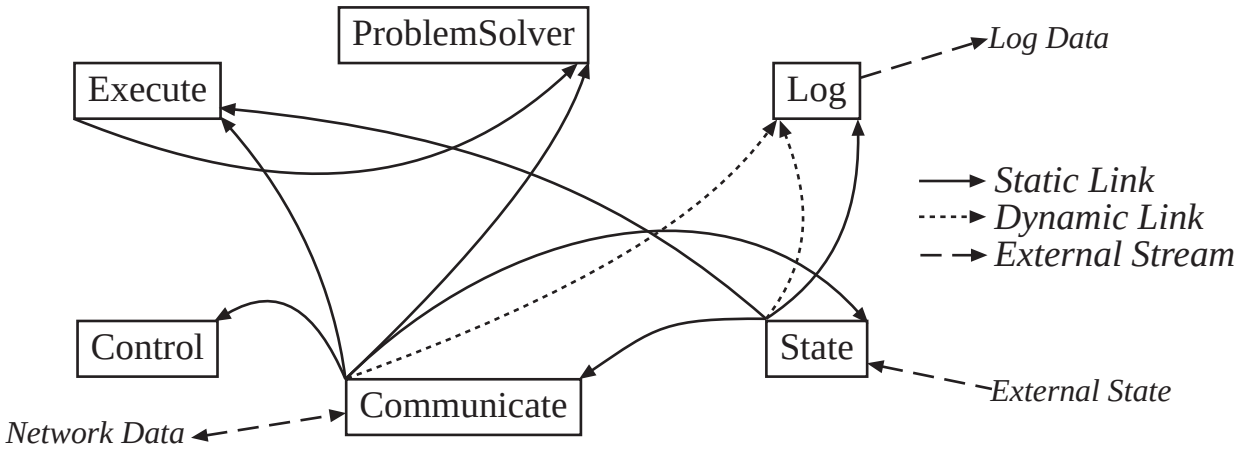

Figure 2: Event stream diagram for the Mass agent

Now that each of the individual components have been examined, it may be useful to see how they are combined and work in conjunction with one another. Using Sun's *beanbox* tool, six components

are first joined together in the applet that acts as a shell holding them together: agent.mass.Control, agent.mass.Communicate, agent.mass.Execute, agent.simplest.Log, agent.mass.State, and agent.mass.ProblemSolver. Since much of their connectivity is represented statically within their source code, very few stream connections need to be added to the group. In particular, only MessageEvent.messageReceived, MessageEvent.messageSent and PropertyEvent.propertyAdded are sent to Log's generic event logging function. Beanbox's property setting interface is also used to set Communicate's host id and port number and Log's logging level. Once completed, beanbox automatically generates and compiles the support and applet code, and then stores the necessary classes within several JAR (Java ARchive) files. The final structural design can be see in Figure 2; static links refer to source code references, dynamic links are those specified within beanbox.

Additional configuration elements, such as the agent's name, location, etc, are placed in a configuration file. A utility was then used to generate the necessary command and command line parameters to correctly instantiate the agent. Execution begins with the *main* method within agent.mass.Control. This function creates the applet context necessary to run the applet class produced by the beanbox tool. After the applet class re-instantiates each of the components, the Control performs system dependency verification. It then initializes each component, during which the bulk of the event stream connections are created. State also gathers the runtime parameters at this time, firing a PropertyEvent for each added variable. After initialization, Control starts each component. All of the components, with the exception of Communicate, are essentially idle at this time. Communicate begins by opening a connection to the simulator, followed by the expected registration handshake which lets the simulator know a compliant agent is connecting.

After the connection is established, the simulator establishes and assigns a unique name to the agent by sending messages the State component can receive and act upon because of its incoming MessageEvent stream. The scenario continues with the simulator sending a time pulse to the agent. Communicate recognizes the pulse message in its preprocessor, and increments the agent's local Clock property. Since no other incoming messages have been queued, Communicate immediately acknowledges the pulse with a message back to the simulator, ending the time segment. The simulator also send notification of the availability of a new TÆMS task structure, a control message which gets fired immediately. ProblemSolver sees the new task structure message and fetches the data from the simulator. After creating a schedule using the task structure, ProblemSolver stores the entire structure in State's table, which causes a PropertyEvent to fire. Execute notices this event, and would halt its currently executing action if it had one.

At the beginning of the second time segment, the simulator sends another pulse message. The Communicator again increments the local clock and fires the pulse event to the other local components. Because a schedule has now been created, Execute responds to this pulse message by fetching the first scheduled task and shipping it off to the simulator. The time segment then ends with Communicate acknowledging the pulse. Several more pulses go by uneventfully, because the agent is waiting for its action to complete. Eventually, the simulator sends back the execution results, which the Execute component retrieves and interprets. It then fires these results to the other components and begins a new task. This process continues until the scenario completes.

This relatively simple example should demonstrate how the behavior of the framework is somewhat different than traditional programming. The event driven mechanism which is used provides the power necessary to exhibit useful behaviors while maintaining clean modular separation.

5 Hypothetical Components

Following up on the examples described in the previous section, I will now give examples of how other hypothetical components could be added to this particular agent. I will concentrate on both the benefits and drawbacks of the framework in these examples, and will leave domain issues to the reader's imagination. The first example is the relatively non-intrusive addition of a learning component. Following this, I will go over how true method execution could be intermingled within a simulated environment.

5.1 Learning

The learning component to be added to the agent will make use of statistical information to help refine the agent's internal model of its behavior, in an attempt to make its decision making process more accurate. To do this, the component will monitor the actions the agent performs, watching such things as quality, cost and duration, and compare them to the expected performance of those actions. It will then use this information to update the model the agent uses when deciding which goals to attempt and how it attempts to achieve them.

To monitor the agent's activity, the learning component first links to the Execute component's ActionEvent stream. Any time an action is completed or aborted, the pertinent information related to that activity will be send over this stream. Using whatever statistical techniques are applicable, the component can use this information to build its own internal view of the characteristics and trends the agent's activities are exhibiting. The component can then retrieve the TÆMS task structure stored in the state table, and compare the expected values to those it observed. If changes are necessary, it will make them, and then store the structure back in the state table.

Only minor modifications to two other components are needed to integrate the new learning component. First, the ProblemSolver needs to watch for changes in the TÆMS task structure. If changes do occur, it needs to verify that it is performing the correct activities, or create a new schedule if this is not true. Second, the Execute component should be modified to be more tolerant of changes to the global task structure. Specifically, it should only cancel and restart action execution if the currently available schedule is changed, rather than when the any part of the task structure changes, as is currently the case.

5.2 Heterogeneous Execution

While it is useful to have a mechanism for simply simulating the execution of a particular task, it is sometimes necessary for an agent to have the actual results one would normally obtain from the action. At the same time, it is still necessary for these actions to be monitored and modified by the simulation controller for the environment to act coherently.

One additional component, able to perform some subset of the available methods, is needed to add this functionality. This component should be able to examine the action characteristics returned by the simulator and produce results matching those characteristics. For instance, if the simulator reports that the results have no quality, the new execution component should recognize this and produce some sort of invalid result. Once this component is designed, it can be placed in-line with the original Execute component, along with some stream redirections, to produce the desired effect.

Using this design, the existing Execute component would still monitor the schedule and send simulated method execution requests to the simulator. Upon receipt of the completion message, though, it would no longer fire an ActionEvent to the original components. Instead, only the new execution component would receive these messages. It would then examine the event and determine if actual results are needed or not. After creating and integrating these results, it would then fire an updated ActionEvent to those components which would normally monitor the stream arising from the Execute component.

This is a fairly simple change to make, but there are some underlying implementation problems which need to be handled. The first is that the static references to the Execute component must be changed in some way. It is true that the new component could be named "RealExecute" or some other phrase that would substring match Execute, but the fact that the original Execute component will still exist in the framework will cause a race condition within the registration process. The names are unordered within the component registry, so requests for "Execute" would not necessarily produce the desired results. There are three viable solutions to the problem: change the original references to explicitly request "RealExecute", generate the links by hand within the beanbox tool, or have a single subclass of Execute rather than have two related instances.

6 Present and Future Directions

The generic Mass agent described in the Example Agent section has been successfully implemented and used with the Mass environment. Four identical agents were used to simulate an example of the multi-agent organization described by the Warren financial portfolio management system [8]. In our simulation, the initial behavior of each agent was governed by the subjective TÆMS task structures seeded to each agent (see section 5.6), which was used by the Design-to-Criteria scheduler to produce a sequence of actions aimed at satisfying a particular goal. Initial survivability scenarios have also been tested, by simulating execution failures which prompt behavioral responses by the agents.

In addition, the agent framework and Mass simulator are also being used in a more complicated environment as part of a group class project. The goal of this project is to develop an intelligent home simulation environment, where agents coordinate over resources and activities in an effort to satisfy both local goals and global constraints. The Mass simulator and TÆMS representation have been updated to support resource models, and the user and agent interfaces improved. Minor additions were made to the agent framework to support the new capabilities of the simulator and facilitate agent construction. A wide range of agents are currently being modeled with the framework as part of this project, including a water heater, dishwasher, air conditioner and coffee pot, to name a few. In each case, the designer only needed to modify the problem solving component of the framework to produce the different behaviors needed by these agents, validating in some sense my prior ease of customization claim. At this time, several agents have been successfully implemented, using new activity scheduling techniques and true agent-to-agent coordination. A new statistics gathering component, reminiscent of the learning module described in section 5.1, has also been designed and seamlessly added to each agent without modification to existing components.

In the coming months, I expect the development of the framework to continue with the design of survivability detection and diagnosis components. One or more coordination components (based on GPGP or using more *ad hoc* techniques) are also planned for development. Modular designs in this

case will theoretically permit us to easily use one or another of these components interchangeably, facilitating the construction and testing process.

7 Conclusion

Component design is an interesting and useful software engineering technique which can greatly facilitate RAD (rapid application development) and code reusability. It is my belief that it is particularly applicable to agent design, because of the functional redundancy which exists in many systems. It can also improve compatibility between existing technologies because of interface conventions, which can assist in the integration process. The end results are programs which have cleaner designs, are better understood and more robust.

The architecture presented in this paper was designed with these goals in mind, and doubtless will continue to evolve as new situations arise. Future work on this project will attempt to provide more control over the event delivery process, including ordering constraints and dynamic response to the event source. Human interface construction may also be formalized or facilitated. It is hoped that as the base of finished components grows, more sophisticated systems can be designed and tested, as reusable components permit the simple construction of heterogeneous but compatible agents.

References

[1] Frank M. Carrano. *Data Abstraction and Problem Solving with C++.* The Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 1995.

[2] Keith S. Decker. Task environment centered simulation. In M. Prietula, K. Carley, and L. Gasser, editors, *Simulating Organizations: Computational Models of Institutions and Groups.* AAAI Press/MIT Press, 1996. Forthcoming.

[3] Alden DeSoto. Using the beans development kit 1.0. http://www.javasoft.com, 1997.

[4] Graham Hamilton (Editor). Sun microsystems java beans api specification. http://java.sun.com/beans, 1997.

[5] Robert Englander. *Developing Java Beans.* O'Reilly & Associates, Inc., 1997.

[6] David Flanagan. *JAVA in a Nutshell.* O'Reilly & Associates, Inc., 2nd edition edition, 1997.

[7] Victor Lesser, Bryan Horling, Frank Klassner, Anita Raja, Thomas Wagner, and Shelley XQ. Zhang. BIG: A resource-bounded information gathering agent. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98)*, July 1998. To appear. See also UMass CS Technical Reports 98-03 and 97-34.

[8] K. Sycara, K. Decker, and M. Williamson. Matchmaking and brokering. In *Proceedings of the Second International Conference on Multi-Agent Systems (ICMAS-96)*, 1996.

[9] Regis Vincent, Bryan Horling, Tom Wagner, and Victor Lesser. Survivability simulator for multi-agent adaptive coordination. In *International Conference on Web-Based Modeling and Simulation*, San Diego, CA, 1998. SCS (eds).

[10] Thomas Wagner, Alan Garvey, and Victor Lesser. Criteria-Directed Heuristic Task Scheduling. *International Journal of Approximate Reasoning, Special Issue on Scheduling*, 1998. To appear. Also available as UMASS CS TR-97-59.

[11] Tom Wagner. Matchmaking: Jil meets tæms. Version June, 13 1997.