A Reusable Component Architecture for Agent Construction *

Bryan C. Horling
Department of Computer Science
University of Massachusetts

UMass Computer Science Technical Report 1998-49

October, 1998

Abstract

A generic, Java-based component architecture (JAF) is proposed as a basis for designing the agents used within multi-agent systems. The goal of this design is to facilitate code reuse and simplify agent construction, by building up a pool of components which can be easily combined in different ways to produce agents with different capabilities. JAF builds upon general component models by adding additional implementation and runtime support designed to produce more consistent and cohesive components. The architecture, based on Sun's Java Beans, is explored, and both domain independent and applied component examples are described in detail.

Keywords: Agent architectures, Agent-based software engineering

1 Overview

Component based architectures [17, 13] have recently begun to receive more attention in the field of software engineering. They attempt to effectively encapsulate the functionality of an object while respecting interface conventions, thereby enabling the creation of stand alone applications by simply plugging together groups of components. 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 we will describe such a component architecture, the Java Agent Framework (JAF), 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, we believe this practice has several weaknesses, both with problems stemming from *ad hoc* agent construction, and in the act of construction itself.

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 [2, 10]. The mechanism which we have implemented, though, is a level of granularity above this. Instead of working with simple objects, agents using our 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-oriented design is particularly applicable to experimental heterogeneous multi-agent systems because changes between agent vari-

*Effort sponsored by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory Air Force Materiel Command, USAF, under agreement number F30602-97-1-0249 and by the National Science Foundation under Grant number IIS-9812755 and number IRI-9523419. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. Disclaimer: The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA), Air Force Research Laboratory, National Science Foundation, or the U.S. Government.

ants can usually be isolated to certain functional areas like an encapsulated planner or problem solver; replaceable modular components can therefore provide an almost plug-and-play form of testbed.

JAF differs from some other agent construction systems [3, 11] in that it provides the designer with a lower level of guidance and support. While a JAF distribution would include several working "agent components," these components would be an example of how to use JAF, rather than being the framework itself. We expect the agent designer using JAF to create a specialized pool of reusable components which meets their needs. While this pool could quite possibly be simple derivations of the included agent components, it need not be restricted to this. In this way JAF provides support for creating multi-agent systems, without limiting the paradigms, techniques or algorithms which actually produce the agents' behavior.

The framework which we have developed is written in Java [8], and builds upon Sun's Java Beans specification [5] 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.

The following sections should give a thorough overview of why JAF was created and how it can be used. We will begin 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 new components will be described. Future improvements and additions will also be covered.

2 Motivation

The motivation for JAF began with the creation of the Multi-Agent Survivability Simulator (Mass) [14]. 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 external 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. The architecture needed for these agents should therefore isolate the agent-dependent behavior logic from the underlying support code which would be common to the entire group. A component-oriented approach satisfies this requirement nicely.

Many component designs, including Java Beans, provide a suitable API for constructing components but leave runtime support to the discretion of the component designer. Our goal is to facilitate the creation of a pool of components, which on one hand have enough flexibility to work effectively and yet are still able to work cohesively as a group. JAF attempts to strike this balance by offering a number of implementation conventions and runtime services, which will be described in the Architecture section below.

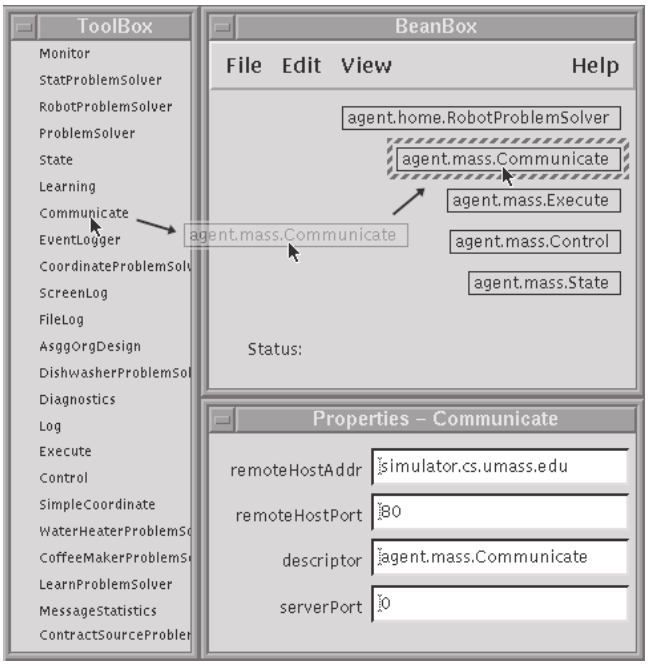

Figure 1: An example agent being constructed.

3 JAF Architecture

JAF was created to provide the designer with a model of construction. It does not provide actual components, instead concentrating on providing direction

and conventions when constructing such components and their higher level organization. In the following section, we will cover the additions JAF has made to generic component-oriented design which support these goals.

JAF is based on the component model architecture Java Beans, which provides behavior and naming specifications to which every component must adhere. Essentially, the Java Beans API is a set of naming conventions which allow both application construction tools and other components to manipulate a component's state and make use of its functionality. Java Beans also provides clients with a notion of component-based event streams, which allow passive, information driven relationships to exist between components.

JAF's grounding in Java Beans supplies it with a component API and event stream model. By themselves, these elements are not sufficient to meet our design goals. The Java Beans framework provides a good component interaction model, but has minimal guidance for generating large component organizations, both in the areas of design and support facilities. To produce a reusable pool of more complex, interdependent components, structure must be added in several areas:

- Components requiring the presence of other components to operate correctly should be able to directly specify these needs.
- These dependencies must be made in loose manner, so a given component can seamlessly be exchanged with a variant.
- Component description matching should be available, enabling components to find their dependencies at runtime.
- A consistent manner of specifying and gaining access to required data and configuration information is needed.
- A more formal execution sequence, with behavior conventions during different stages, would facilitate component interactions.

How each of these requirements is addressed in the JAF architecture is covered in the following sections.

3.1 Component Dependencies

Most of the components used within an agent rely on the services provided by other components. A problem solving component directing the activity of an agent, for instance, must have access to communication and execution facilities if it is to operate correctly. The exact nature of these facilities, however, varies depending on the agent's environment. As an example, the problem solver, which normally operates in a real world environment, would use different components when it is running as part of a Mass-based simulation. These different components would need to be aware of the simulation controller, and interact with it as needed. Compounding this problem, it is desirable to have such a change be transparent; no changes to the problem solver should be needed regardless of what variety of communication component is actually present. To resolve this tension, JAF includes facilities for loosely specifying and finding component dependencies.

Upon initialization, each JAF component will register its dependency list, containing descriptions of components which must be present for the registering component to operate. JAF then uses this description to find and return the most closely matching component. For example, if the problem solver specified a "Communicate" dependency, it would match both a real "Communicate" or a "Mass Communicate" component, allowing the designer to simply plug the correct communication object into the agent without modifying the problem solving code. Looking at Figure 1, one can see that the Communicate component has a descriptor property, a characteristic all JAF components have. This descriptor string is what is compared to a dependency description when searching for matches. This mechanism of statically defining dependencies while dynamically using the closest match at runtime is a convenient way of making use of the capabilities provided by different components. This ability also allows components to generate event stream connections at runtime, thereby allowing both passive (event stream-based) and active (method-based) inter-component connections to form dynamically.

3.2 Information Dependencies

Individual components also have information dependencies, ranging from simple behavior altering flags to complete external data files. Similar to the component dependencies described above, JAF supplies convenient mechanisms for describing, storing and gaining access to data dependencies. As components are constructed at runtime, each may register its local data dependencies, providing the desired data's name, type and description. Armed with this infor-

mation, JAF will search for the data in several places and read in the relevant information, convert it to the desired type, and store it for later retrieval. Components themselves may also write to and monitor changes to this storage table, which allows it to serve both as a central shared memory area and source of reactive control. JAF also provides easy access to an external configuration archive, where large free-form textual or binary objects may be stored.

The object which provides the data services described above, named State, is itself a component within the agent. Because of this, we may take advantage of JAF by replacing it with an equivalent component, if needed. This allows us, for instance, to seamlessly replace the generic State component with a specialized "Mass State" component (seen in Figure 1) which is able to send and receive data information with the Mass simulation controller.

3.3 Control Flow

Although hierarchical organizations are possible, most of the components built with JAF thus far are organized in a peer-to-peer manner. If no centralized notion of control existed, it would be difficult for components to actively use one another because of race conditions caused by different states of execution. If one component attempted to call a method in another before that component had performed some necessary setup, unexpected results may occur.

To strengthen the flow of control within the agent, JAF provides components with a phased view of execution, and conventions associated with each phase describing what activities are permitted. Execution begins with the initial component construction all objects go through when they are created. During construction, each component is expected to register both its component and informational dependencies. Before agent execution begins in earnest, a check is made to ensure that all component dependencies are satisfiable. This check is then followed by walking the installed components through several defined phases.

During the first phase, initialization, the only guarantee made about the system state at is that all other components have registered successfully. Because of this, initialization typically will consist of each component setting up their internal state, obtaining references to component dependencies, and forming event stream connections to other components.

During the next phase, each component is started. The primary activity of the component should begin during this stage, and significant inter-component interactions can now take place. Examples of such activity include a communications module setting up server capabilities, or a problem solver analyzing which goals to achieve. Pulse delivery coupled with event reciept reactions forms the backbone of execution flow within a typical agent.

Our current control paradigm assumes a discrete view of time, so the main execution phase consists of each component being periodically pulsed. During each pulse, a component may perform whatever actions it deems reasonable for unit time. From a simulation standpoint, this has the advantageous side effect of being synchronizable from a centralized source (such as the simulation controller). Analogous to the Mass-specific State component described above, there is also a Mass-specific Control component which does just this, by interpreting pulse messages arriving from the simulation controller.

JAF also offers the ability to reset components, halt agent execution and control multiple threads of execution.

3.4 Communicate Example

```
class Communicate extends AgentComponent implements PropertyEventListener {
 // Normal constructor
 public Communicate() {
  State.addParameterInfo("Host", "String", "Host IP"); // Data dependency
  addDependency("State");                              // Component dependency
 }

 // Called during initialization phase
 public void init() {
  state = (State)State.findComponent("State"); // Find State component
  state.addPropertyEventListener(this);        // Listen to property events
 }

 // Called during begin phase
 public void begin() { // Open connections here }

 // Called periodically during main execution phase
 public void pulse() { // Not used by generic Communicate }

 // Used to send messages to external hosts
 public synchronized boolean sendMessage(Message m) { ... }
}
```

Figure 2: Communicate component sample code

Figure 2 shows portions of an example component which make use of JAF's capabilities. This should give the reader an sense of how the concepts in the preceding sections are used in practice. This particular component, named Communicate, is responsible for handling all network messaging services within the agent. In the constructor, one can see both a data and component dependency being registered. Following this are the three methods (init, begin and pulse) called by the Control component during the respective phases of execution. In init can be seen a call to the findComponent method, which searches for components matching the given description, followed by a

call which dynamically registers Communicate as listener to State's property event stream. Also shown is sendMessage, the interface used by other components to send messages to other hosts.

4 An Example Agent

In this section we will explore the construction of a generic agent and its components, using the framework described above. Each component will be described both in terms of its behavior and capabilities, as well as interactions with other components, which should give the reader a notion of how components are designed and operate in practice. Interesting cases where JAF capabilities are used will be noted. To underscore the reusability of components, the specialized Mass agent will also be discussed (see Figure 1), with attention paid to how existing generic components were augmented to work within our simulation environment.

4.1 Design

The generic agent consists of 6 components: Communicate, Control, Execute, Log, ProblemSolver, and State. The ProblemSolver component encapsulates the behavior of the agent, making use of Communicate and Execute to perform messaging and action tasks, respectively. As discussed previously, Control directs the initialization and activity of each component, and State provides access to required component references and data. Log is used by all components, serving as a central location for free-form text and event logging services. The sections below will give a more complete view of each component, followed by a trace through the internal activities of the agent.

4.1.1 Communicate

The Communicate component (see also Section 3.4) is responsible for handling the network message activity within the agent. Communicate starts by using JAF's data services to register a need for information, notably a host name and port number (refer to the Properties box in Figure 1). When the component is started, it opens up a connection to the remote host (if specified), and listens for incoming message on the numbered port. As messages arrive, they are decoded and sent to any listeners registered to Communicate's event stream. Communicate also supplies a sendMessage method, which other components may use to send messages. Components wishing to monitor the receipt or delivery of messages need only listen to the Communicate's Message event stream.

The Mass derivation of Communicate adds support for the simulation environment. Minor changes were made to support KQML [7] encoding of information and watching for simulation-specific control messages. The pulse phase of execution, unused in the generic Communicate, is used to deliver queued message events to the components, since no activity should take place until the simulator wakes up the agent with a pulse message.

4.1.2 Control

The generic Control component is identical to the one described in the architecture section. It is responsible for checking component dependencies, initializing and pulsing each component, and correctly resetting and quitting the agent when necessary.

To allow the simulation controller to affect the local execution within the agent, the Mass Control component was modified to enable it to listen to Communicate's message event stream. It begins by registering a component dependency on Communicate. During initialization, it uses JAF's component services to find the Communicate instance present in the agent, and connect to its event stream. During runtime, it then monitors this stream for pulse, reset and disconnect commands arriving from the centralized controller.

4.1.3 Execute

The Execute component is responsible for performing the large-grained actions associated with the agent, which could range from direct responses to queries to simply performing tasks laid out in a global schedule. Because the specific mechanism for executing tasks is usually domain dependent, the generic Execute component functions more as a template, providing mechanisms for starting, tracking and aborting actions, but not actually performing them. Execute supports an Action event stream, which carries information about actions which are started, completed or aborted.

The Mass Execute component builds on the simplest template described above by adding the functionality needed to simulate execution. Execute handles agent actions by sending execution requests to the simulation controller, which determines the specific characteristics of the simulated task. The Execute component then makes use of the Message event

stream from Communicate to watch for the controller's notification of task completion.

4.1.4 Log

The Log component is a simple logging class, which provides generic methods to facilitate both textual 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.

4.1.5 State

The State component is identical to that described in the architecture section. It supports data retrieval and storage, and supplies an event stream allowing components to monitor additions, removals or changes to local data.

The Mass State component adds support for remote property queries and changes by monitoring Communicate's Message event stream, allowing for greater agent control by the simulation controller.

4.1.6 ProblemSolver

In a typical agent, the problem solver is the domain expert and thus acts as an instigator to other components. The problem solver can be any type of domain expert, e.g., generative planner, process program, expert system, etc. It is generally responsible for discovering and evaluating problems or goals, and determining how best they can be solved or achieved. Within the generic 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[1]. The job of creating task structures for individual agents has been delegated to a task generator [9], housed either within the centralized simulation controller or another component local to the agent. When a simulation scenario begins, this generator is responsible for creating the task structure, which the agent then uses to schedules its activity by.

The Mass problem solving component, then, is responsible for using the generator to construct a task structure. Once the structure has been obtained, it can be run through the Design-to-Criteria scheduling system [15], which attempts to produce an execution schedule satisfying the agent's goal criteria. More sophisticated behaviors can be incorporated into the agent by replacing this component. As an example, in following section a specialized RobotProblemSolver component will replace the generic problem solver.

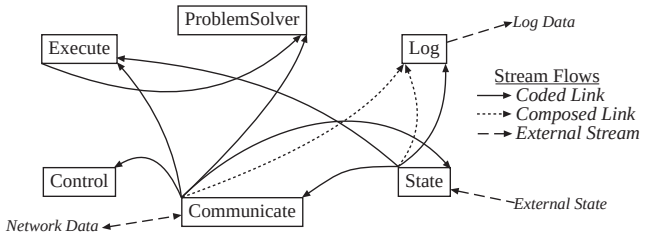

Figure 3: Event stream diagram for the Mass agent

4.2 Integration

Now that each of the individual components have been examined, it may be useful to see how they are combined and work with one another. The following example shows how a Mass-compliant agent, the mobile robot, is constructed and run.

Using an appropriate Java Beans manipulation tool, the components are first joined together in an applet, which acts as a shell holding them together (see Figure 1). Within the tool, event streams can be sent to individual listener components, but since much of their connectivity is represented statically within their source code, very few such connections need to be added to the group. In particular, only message and property events are sent to Log's generic event logging function to facilitate runtime troubleshooting (shown as the Composed Links in Figure 3).

Additional configuration elements, such as the agent's name, location, etc, are placed in a configuration file. A utility is then used to generate the necessary command and command line parameters to correctly instantiate the agent. After the agent has started, Control performs component dependency verification. It then initializes each component, during which the bulk of the event stream connections are created (shown as Coded Links in Figure 3). State also gathers the runtime parameters at this time, firing a property event for each added variable. After initialization, Control starts each component. All of the components, with the exception of Communicate and the robot problem solver, are essentially idle at this time. Communicate begins by opening a connection to the simulator, followed by the expected registration handshake

[1] Though we often add a component to the domain expert problem solver that converts its internal structures to TÆMS task models, thus enabling the problem solver to interface to the rest of the tools in the same fashion as the generic problem solver.

which lets the simulator know a compliant agent is connecting. Other information, such as agent name, and random number seed are also exchanged at this time. The problem solver uses a local scripted task structure generator to produce a task structure describing its possible goals and methods of achieving them. In this case, it describes what activities the robot is able to perform, and lays out what routes in the house must be followed to perform them.

The scenario continues with the simulator sending a time pulse to the agent. Communicate recognizes the pulse message in its preprocessor, and, using State, increments the agent's local Time property. Control also recognizes the pulse message, and reacts by sending a pulse to each of the local components. Nothing happens in the agent at this time because no contracts have been received. Since no other incoming messages have been queued, Communicate ends the time segment by acknowledging the pulse with a message back to the simulator.

When a contract message arrives at the agent, the Communicate component sends a message event to its listeners, which includes the robot problem solver. After determining what kind of message it is, the problem solver can then parse the message, analyze the contract, and possibly use Communicate to reply with a bid. If such a bid is later accepted, the ProblemSolver will create a schedule that Execute can use to perform actions. Since the ProblemSolver also listens to Execute's action event stream, it can easily monitor the progress of the schedule, and make adjustments as necessary.

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

4.3 Customization

The simple agent described above can and has been augmented in different ways to provide new capabilities. The most straightforward change is to replace the ProblemSolver component with a more domain dependent variant. For instance, a coffee maker problem solver could have knowledge of its resource requirements, and attempt to coordinate over them before beginning its execution. A mobile robot problem solver could have knowledge of its location and then use the scheduler to analyze its proposed task schedule as it moves about its environment (such an agent is seen in Figure 1).

Other less-intrusive additions also exist when taking advantage of the naturally decoupled nature of the components. For instance, using a graphical interface, an evaluation criteria component could obtain the user's schedule evaluation preferences at runtime and attach them to the task structure before scheduling occurs. In this case, no modifications to existing components would be necessary for the agent to exhibit significantly different behaviors. Simple statistics gathering components in an agent could be attached to various streams to monitor the type and content of events as they are fired. Such information can then be used by yet another component to perform runtime behavior analysis.

A diagnosis module [1] could also be added to the agent. In this case, the component could monitor the behavior of other components and even other agents, using this information to support diagnoses. Some of these diagnoses could then be used by the agent's local task generator to change the behavior of the agent. With the aid of the Communication module, distributed diagnosis could also take place, as diagnosis components in different agents collaborate to recognize a particular problem.

5 Current & Future Work

To date, over 27 JAF components have been designed and implemented. The Mass agent described in Section 3 has been successfully implemented and used within the Mass environment. Four agents using a simple problem solver were also used to simulate an example of the multi-agent organization described by the Warren financial portfolio management system [12]. Initial survivability scenarios have been tested, by simulating execution and resource failures which prompt behavioral responses by the agents.

In addition, the agent framework and Mass simulator are also being used in a more complicated multi-agent environment [6]. 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. A wide range of agents are currently being modeled with the framework as part of this project, including a water heater, dishwasher, mobile robot, 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 our prior ease of customization claim.

In the coming months, we expect the development of the framework to continue with the design of survivability detection and diagnosis components. One or more coordination components based on $GPGP^2$ [16, 4] are also under development.

6 Conclusion

The agent framework described in the paper, JAF, has been developed as a new model for constructing agents. JAF places particular emphasis on encapsulation and modularity, in an attempt to promote and facilitate code reuse which in turn can make agents both more compatible and sophisticated. While these attributes are arguably desired in any application, we feel that they are particularly useful within an multi-agent testbeds, because these systems often employ sets of heterogeneous agents with common subsystems and interfaces.

The framework itself provides support mechanisms for control and data flow, in addition to component design guidelines. Agents using JAF are composed of a number of such components, which interact with one another using both passive event stream monitoring and active method invocation. This type of organization allows for a wide array of implementations, while maintaining a consistent overall design.

Thanks to Victor Lesser, Régis Vincent and Tom Wagner for their helpful critiques of this paper.

References

[1] Ana L.C. Bazzan, Victor Lesser, and Ping Xuan. Adapting an Organization Design through Domain-Independent Diagnosis. Computer Science Technical Report TR-98-014, University of Massachusetts at Amherst, February 1998.

[2] T. Biggerstaff and C. Richter. Reusability framework, assessment, and directions. In T.J. Biggerstaff and A.J. Perlis, editors, *Frontier Series: Software reusability: Volume I - Concepts and Models*, chapter 1, pages 1–17. ACM Press New York, 1989.

[3] Deepika Chauhan and Albert D. Baker. Jafmas: A multiagent application development system. In *Proceedings of Second International Autonomous Agents*, pages 100–107, 1998.

[4] E.H. Durfee and V.R. Lesser. Partial global planning: A coordination framework for distributed hypothesis formation. *IEEE Transactions on Systems, Man, and Cybernetics*, 21(5):1167–1183, September 1991.

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

[6] V. Lesser et al. The UMASS intelligent home project. Submitted to Agents99.

[7] T. Finin, R. Fritzson, D. McKay, and R. McEntire. KQML as an agent communication language. In *Proceedings of the Third International Conference on Information and Knowledge Management CIKM'94*. ACM Press, November 1994.

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

[9] Prasad Nagendra, K. M.V., Decker, A. Garvey, and V. Lesser. Exploring organizational designs with taems: A case study of distributed data processing. In *Second International Conference on Multi-Agent Systems*, pages 283–290, 1996.

[10] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modelling and Design*. Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[11] Yoav Shoham. Agent0: A simple agent language and its interpreter. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, pages 704–709, July 1991.

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

[13] J. van den Elst, F. van Harmelen, and M. Thonnat. Modelling software components for reuse. In *Seventh International Conference on Software Engineering and Knowledge Engineering*, page not yet known. Knowledge Systems Institute, June 1995.

[14] 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).

[15] Thomas Wagner, Alan Garvey, and Victor Lesser. Criteria-Directed Heuristic Task Scheduling. *International Journal of Approximate Reasoning, Special Issue on Scheduling*, 19:91–118, 1998. A version also available as UMASS CS TR-97-59.

[16] Thomas Wagner, Victor Lesser, Brett Benyo, Anita Raja, Ping Xuan, and Shelly XQ Zhang. Gpgp2: Improvement through divide and conquer. Working document, 1998.

[17] B.W. Weide, W.F. Ogden, and S.H. Zweben. Reusable software components. In *Advances in computers, vol. 33*. Academic Press, 1991. ISBN 0-12-012133-6.