Agent Behavior Architectures
A MAS Framework Comparison

Steven P. Fonseca
UC Santa Cruz / HP Labs
Baskin Engineering Building
Santa Cruz, CA 95064
1-650-236-2949
fonseca@cse.ucsc.edu

Martin L. Griss
Hewlett-Packard Labs
1501 Page Mill Road, 1U-14
Palo Alto, CA 94304
1-650-857-8715
martin_griss@hp.com

Reed Letsinger
Hewlett-Packard Labs
1501 Page Mill Road, 1U-16
Palo Alto, CA 94304
1-650-857-5974
reed_letsinger@hp.com

ABSTRACT
Advances in agent technology depend on improving frameworks for building and supporting agent societies. Experience suggests that first generation multi-agent systems fall short of providing a rapid prototyping development environment for the systematic construction and deployment of agent-oriented applications. While at least sixty [13] different agent systems have been implemented, few efforts have been made to use them as case studies for building second-generation multi-agent systems. We propose a refactoring of both architecture and implementation across the FIPAOS, JADE, and Zeus open-source agent frameworks to produce a new multi-agent system framework called MAS2. The first step is to extract reusable design elements for MAS2 beginning with the agent behavior subsystem. FIPAOS, JADE, and Zeus all have a core behavior subsystem that includes an execution process, ACL message interface, agent behavior engine, and corresponding primitive processing objects. These mechanisms are evaluated in this paper in the context of building a meeting scheduling protocol from which useful architectural elements and implementation techniques are identified.

Categories and Subject Descriptors
D.2.2 [**Software Engineering**]: Design Tools and Techniques – *modules and interfaces, object-oriented design methods, software libraries, state diagrams.*

General Terms
Design, Languages

Keywords
Agent behavior, multi-agent system, FIPAOS, JADE, ZEUS.

1. INTRODUCTION
MAS frameworks attempt to provide programmers with reusable agent-oriented classes which share useful relationships. To build a framework, the agent domain must be decomposed and programming abstractions identified. This process is called agent-oriented software engineering and programming. An introduction to these topics is provided in the next two sections after which a discussion of the research questions is given.

1.1 Agent-Oriented Programming
Agent-oriented programming involves using an agent as the main unit of encapsulation. In this paradigm [12] "we program an agent with the help of high-level elements such as goals, choices, skills, beliefs, and so on, and that the types of message that agent exchanges with each other also refer to high-level communication mechanisms in defining messages about data, requests, offers, promises, refusals, acceptances and so on." In particular, multi-agent systems (groups of agents working together to achieve some purpose) are critical for many large-scale e-commerce and intelligent assistant systems, including negotiation, brokering, contract formation, selling, scheduling, etc. The group of agents can be a static or dynamic grouping, or a combination of static and dynamic. The communication can be between people and agents, or between agent and agent, or a mix.

Before effective agent-oriented solutions are available, agent-oriented engineering and process methodologies must be matured. This lesson was learned after building a mobile shopper agent-based e-commerce framework and an agent-based meeting scheduler at Hewlett-Packard Labs. Focusing on the engineering of multi-agent systems, the programming techniques used to handle distributed system complexity is essentially the same as dealing with any type of sophisticated software; decomposition, abstraction, and organization are key [3,8]. In a very general sense, the purpose of our research is to take these three mechanisms and apply them to the agent-oriented paradigm. This attempts to answer a "fundamental question" of agent-oriented software engineering [9]: "What are the essential concepts and notions of agent-based computing." Jennings identifies agents, agent organizations, interactions, and environment as possible concepts. While focusing on abstracting and decomposing agent behavior rather than studying agent or agent-society level issues, we build on his ideas introducing alternate vocabulary.

Like the transition from functional to object-oriented programming, moving from object-oriented to agent-oriented programming involves building on top of and re-encapsulating units of computation. The additional layers of abstraction introduced by this paradigm appearing in Table 1 are named by their encapsulation component. The crossover from the object to agent layer begins at the multi-agent system level. Although multi-agent system frameworks can be thought of as being parallel to rather than extending from traditional object-oriented frameworks, they are placed at a higher level of abstraction because they typically utilize object-oriented API's. Well-written

Table 1. Agent-oriented programming abstraction layers

Agent-Oriented Programming Abstraction Layers		
Abstraction Level	**Description**	**Example**
Agent Environment	Collection of agent societies	FIPA infrastructure agents and application agents
Agent Society	Collection of agents	FIPA infrastructure agents
Agent	Collection of executing roles	Meeting requesting agent
Agent Role	A well-defined high-level agent action	Meeting participant protocol
Agent Infrastructure – MAS Framework	API for low-level agent action and society infrastructure action	FIPAOS, JADE, Zeus
Application Framework or API	OO API for language extension and raising abstraction	RMI, MFC
Language	Core software foundation	Java

MAS frameworks recast object calls using agent-oriented concepts. JADE, the Java Agent Development Environment [2], is an example multi-agent system framework that provides the ability to communicate with FIPA compliant society infrastructure agents.

An agent API provides the fundamental capabilities that an agent must possess to participate in the default society chosen by the MAS developers. JADE agents are given the ability to communicate with FIPA compliant society infrastructure agents. The agent API also defines interfaces to the agent subsystems. The number and types of subsystems provided by a MAS differ from implementation to implementation – domain dependent agent requirements often dictate design choices. The trend is for MAS's to minimally provide some level of conversation management, rule-based behavior, and function-based behavior subsystems.

As was stated, most MAS frameworks provide an agent behavior subsystem. These subsystems are often called "engines." Whether rule-based or object-based, the function of a behavior engine is to provide built-in support for programming agent behavior. Regardless of the application domain, there is a set of actions or utilities that are commonly needed by any agent interacting within a society. Much like typical libraries, factoring these utilities into MAS support classes alleviates agent programmers from reinventing behavior. Message storage is one example facility provided by the Zeus coordination engine [4]. As is the case with Zeus, a behavior engine can offer domain specific functionality (buy, sell) in addition to general-purpose agent behavior support. At a minimum, engines need to offer general-purpose support because the intention is for programmers to extend, customize, and tailor agent behavior to the problem domain. Engines typically provide a pseudo-concurrent mechanism enabling multi-tasked agent behavior execution. This offers a single-threaded multi-tasking solution.

JADE, FIPAOS, and ZEUS behavior engines adopt architecture similar to Container-Based Component Management that is used by the J2EE environment. Containers or engines provide guaranteed services to their respective components. For example, a web container provides request services to its Servlet components. Analogously, an agent engine container might provide a new dialogue convenience service to its agent role component.

An agent role is meant to represent a well-defined and encapsulated unit of high-level agent behavior. An agent role can denote the same behavior expressed in defining a protocol role. Alternatively, it can represent some other action an agent performs that may not involve communicating with the society. An agent role is composed of primitive operations from the MAS framework abstraction layer and can also encapsulate other lower level agent roles. In FIPAOS, JADE and ZEUS, primitive operations are contained in objects implementing an interface that enables the behavior engine to manage and execute them.

Depending on the implementation, the internals of an agent role can range from loosely to highly structured and consequently dictate the same degree of coupling with the behavior engine. Loosely coupled agent roles afford the programmer a high degree of flexibility but don't take advantage of factoring out reusable code across agent roles. Highly structured agent roles are more constraining but can take advantage of built-in engine functionality and may offer programmers a solution template for writing agent behavior. Two concerns emerge from these observations. First, the success of using a highly structured agent role is dependent upon how easily agent behavior can be mapped to the internal constructs. The solution model must align with the agent role programming model. Second, there exists a tension between flexibility and reusability that must be tuned through experience.

Agent roles can be rule-based or object-based; all MAS frameworks support both. An interesting encapsulation implication for rule-based behaviors is if all rules appear at the same level, a system rapidly becomes unwieldy as the number of rules increase. Instead, programmers think about "rulesets" or "rule modules", and use a top-level set of rules to select the active rule model.

1.2 MAS Frameworks

FIPAOS, JADE, and Zeus are all open source Java based MAS frameworks implementing, to varying degrees, FIPA (Foundation for Intelligent Physical Agents) compliant agent systems. The features these frameworks provide are similar as they share the common goal of providing a code base for developing intelligent, distributed, and autonomous software, using agents as the unit of encapsulation. As will be apparent, the major subsystems of these frameworks are the same. What varies is their high-level

architecture and, to an even greater extent, their implementations. An overview of each framework is offered here to establish the context for comparing their support of function and state based programming of agent behavior discussed in Section 3.

1.2.1 FIPAOS

Nortel Networks developed the FIPAOS MAS framework with the intent of providing a platform whose architecture emphasized ease of extension, modularity, and therefore openness [11]. As with JADE and Zeus, FIPAOS source code is freely available under the terms of a license prohibiting commercial exploitation. FIPAOS, in accord with numerous FIPA specifications [6], provides support for agent management, ACL message transmission and reception, and protocol adherence. It includes an agent shell that serves as the foundation for building customized agents, a task manager for constructing agents from primitive work units called tasks, a conversation manager that ensures protocol compliance while also providing conversation utilities for tasks, and a message transport service for sending and receiving messages. Factories are provided for databases and parsers. FIPAOS can also be used with the JESS [7] rule-based system. These subsystems are connected by defined interfaces enabling component interchange. FIPAOS includes runtime tools for managing agents, viewing thread pools, and monitoring task execution. A task generator tools is also included that takes a protocol declaration and generates all of the classes required to implement each role. FIPAOS provides a name server, directory facilitator, and agent communication channel agents. An input-output agent with a GUI interface allows ACL messages to be sent as specified manually by the user.

1.2.2 JADE

Just as FIPAOS, the JADE MAS framework has implemented all of the mandatory components of the FIPA specifications. Developed at CSLET, the primary objective of this framework is to make it easier to program multi-agent societies whose agents interact in compliance with FIPA. JADE provides [2] the same FIPA infrastructure agents as FIPAOS, and a container paradigm for associating JVM's, agents, and hosts. Also included is an agent foundation class for writing customized agents, a library of protocol skeletons for tailoring agent conversation, a behavior hierarchy for programming function-based agent execution, an interface for using the JESS rule-based system, and a suite of development tools. The tools provided include runtime agent management, directory facilitation monitoring and editing, message exchange debugging, agent life-cycle control, and a conversation monitoring tool that draws a sequence diagram of agent interaction.

1.2.3 Zeus

Zeus was developed at the British Telecom Labs [4]. It adopts a layered approach to agents. Basic agents are able to follow an agent communication protocol with other agents in the community. They have the ability to plan sequences of steps needed to accomplish a goal, to monitor the execution of plans, and backtrack when necessary. Also, these agents can be supplied with forward-chaining rules that respond to perceived changes in the environment. Zeus agents are goal directed. The MAS framework provides a comprehensive suite of monitoring tools at the agent and society level. Also provided is an agent generator tool for configuring agent subcomponents and internal recourses at compile time.

1.3 Research Questions

The general purpose of agent-oriented programming is to raise the level of abstraction of components composing a software system and minimize the coupling (and increasing the autonomy) that exists between these distributed pieces of software. The engineering of agent systems further requires identifying reusable agent-specific and domain-specific abstractions, correctly partitioning agent code into reusable components, tool support for agent construction and monitoring, and maturation of agent frameworks.

We focus on evolving MAS frameworks to support rapid programming of agent behavior. The following questions are of importance: 1) what is a good algorithm and implementation for routing incoming messages to the correct execution unit? 2) What amount of intelligence is useful in dispatching messages and how is this intelligence programmed? 3) What high-level message management policies should the MAS platform provide to an agent by default? 4) What built-in support, if any, should be provided by a platform to support state-based agent composition? 5) How can a MAS platform help rapidly construct agents?

Additional questions concerning agent behavior include: 1) How can agent behavior be decomposed into high-level actions? 2) How can an agent be constructed from reusable components? 3) What are these components and what interface do they share? 4) What architectures make it easier to understand and maintain agent behavior? 5) Is state-based agent behavior a useful programming paradigm? 6) How are the processing steps an agent executes partitioned into states and their transitions?

2. EXPERIMENT DESCRIPTION

This section gives a brief description of the test application that was written to evaluate the agent behavior programming support provided by the FIPAOS, JADE, and Zeus MAS frameworks. The experiment is classified according to criteria defined by Basilli [1] to establish the context for evaluating the results of this study. Following this is a description of the meeting scheduling protocol and corresponding design diagrams.

2.1 Experiment Classification

The motivation of this study is to assess and improve current MAS framework behavior execution implementations to develop a second-generation multi-agent system. An experienced Java programmer, from the perspective of both the framework developer and user, will evaluate FIPAOS, JADE, and Zeus in the context of building a meeting scheduling protocol. The scope of the work is considered a replicated project as the same meeting scheduling protocol is implemented using three different MAS frameworks.

2.2 Problem Description

Motivation for the experimental application was taken from a multi-agent meeting-scheduling prototype developed at Hewlett-Packard Labs. In this scenario an agent initiates a request for a meeting to be scheduled. The meeting agent receives this request, which includes a list of invitees, and asks the desired participants to provide their availability. After receiving their responses, the meeting agent determines the subset of available times, considers additional meeting constraints defined by the requester, and then sends a meeting proposal message to the participants. This message contains such information as the meeting time, location, and available room resources. Participants are then expected to accept or reject the meeting attributes. If the meeting agent decides to hold the meeting, it informs the participants who are expected to confirm their attendance. The meeting agent also tells the meeting requester that a meeting has been successfully scheduled. The meeting protocol just described includes a negotiation loop enabling the meeting attributes to be adjusted until the meeting agent is satisfied with the expected participation.

2.3 Protocol and State Machine Design

Complementing the textual description of the experimental application is a sequence diagram of the meeting protocol and a

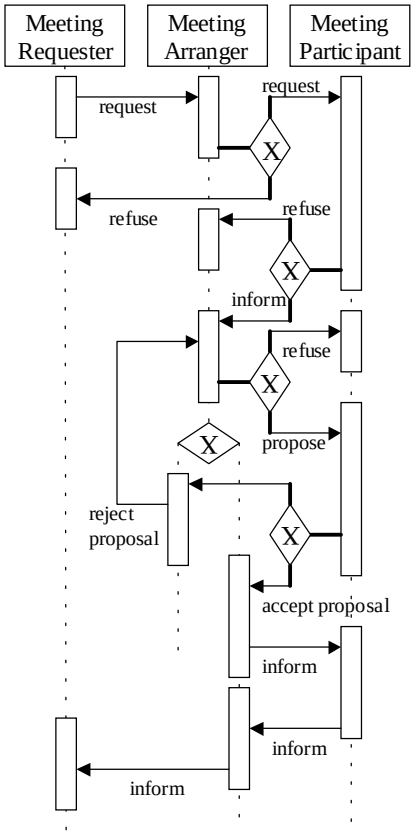

Figure 1. AUML sequence diagram of meeting scheduling protocol

sample state machine chart for the Participant role. Notice that FIPA verbs [5] were used to define the exchanged message types and AUML [10] was used to model the conversation. While the sequence diagram in Figure 1 provides a society level view of interaction, the state machine chart in Figure 2 illustrates the internal processing of agents performing a meeting agent role.

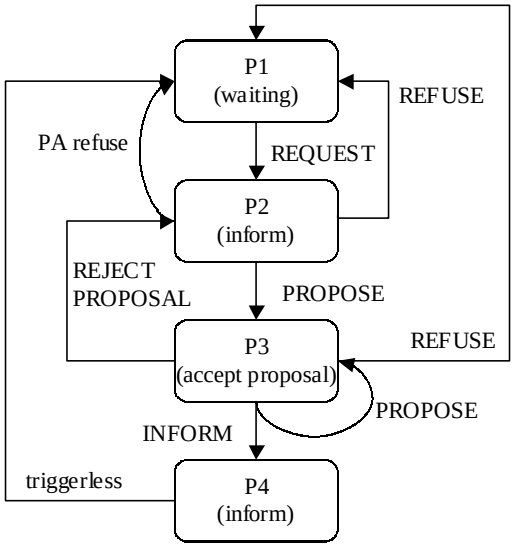

Figure 2. State chart diagram of meeting participant role

This view is particularly useful for agents implemented using state-oriented programming.

3. BEHAVIOR ARCHITECTURES

The meeting scheduling protocol described above was implemented using the FIPAOS, JADE, and Zeus frameworks. Each provides a function-based behavior execution mechanism that allows programmers to construct agents from primitive processing elements. A discussion of the architecture and implementation for each framework is provided and a few platform comparisons are made. Further analysis is found in the subsequent section in which the elements common to these frameworks are evaluated.

3.1 FIPAOS

The primary components of the FIPAOS behavior execution mechanism are a conversation manager, conversation, task manager, task, task state, and task event. These components, and additionally the agent shell and message transport service (MTS), are connected via well-defined Java interfaces to assist programmers with sending, receiving, and processing ACL messages.

The conversation manager, implementing the message receiver interface, is forwarded incoming messages from the agent society by the MTS. The conversation id is checked to determine if the message is associated with an outstanding conversation. If it is, the corresponding conversation object is retrieved, protocol compliance is checked if the conversation follows one, and the conversation object stores a copy of the message. If the incoming message indicates the beginning of a new conversation, a new conversation object is instantiated, compliance is checked if a protocol is specified, and the message is added to the conversation. The final responsibility of the conversation manager is to add the new or updated conversation object to a queue that is monitored for objects implementing the conversation listener interface.

The task manager of the behavior execution subsystem implements the conversation listener interface. When it receives a

conversation object it inspects the conversation id to determine if a task has already been associated with the conversation. If it has, a task event containing the conversation object is created and added to the task. If no task is associated with the incoming conversation, then the idle task is given the event task object. An idle task must be written to handle incoming new conversations where the receiving agent is not the initiator.

Once a task event is added to the appropriate task, the task manager informs the task manager listener that a new task event was generated. The task manager listener replies by telling the task manager where to place the task in the queue of tasks waiting to execute. The task manager executes tasks in appropriate order. The queue is checked for tasks, all events for each task are handled, and then a second elapses before the next execution cycle begins.

Task execution involves decoding the type of event requiring service and calling the appropriate task method. FIPAOS defines the following event types: Initialized event, conversation update event, child timeout event, child failure event, and child done with result event. Naming conventions are used to map incoming events to the correct task method. For example, a conversation update event with a conversation object that contains a request performative would call the task method named handlerequest(Conversation). Similarly, when a child done event is serviced, its parent task is expected to have a method whose name is the name of the child task with 'done' appended to the end.

A parent-child task relationship is supported by FIPAOS. When a child task completes the system generates a child done event that triggers a callback method to execute, as previously described. Built-in support for sharing of data between parent and child tasks is provided by get and set API calls to manage a context object. There is also support for a child to return a value to the parent after execution.

In concluding this discussion on the FIPAOS behavior execution subsystem, several additional features require explanation. First, the message sending mechanism is analogous to the previously discussed receiving mechanism both in mechanics and in interface usage. Second, FIPAOS provides classes for all of the FIPA protocols. The skeleton of execution code for these protocols is not provided. Instead, the classes define the legal sequence of message exchanges for a conversation. A task generation tools is included with FIPAOS that takes this protocol definition and generates tasks with appropriate method stubs. Finally, as with JADE, FIPAOS was designed to work with the JESS rule-based system.

3.2 JADE

Similar to Zeus, the JADE MAS framework provides a behavior engine for programming agent behavior using homogeneous processing elements that are executed in pseudo-concurrent fashion. JADE, however, defines a hierarchy of behavior classes implementing common execution patterns rather than tailoring the engine to process state machines. New versions of JADE do offer state machine execution support on top of their existing infrastructure. But this mechanism is not nearly as well developed as in Zeus. The JADE engine processes behaviors as opposed to states within a given state machine behavior.

Behaviour, the root class of the behavior hierarchy, defines several core methods whose semantics are similar to methods of the UML state concept. The onStart and onEnd methods are executed upon entry and exit of a behavior. An action method embodies the core code for the behavior. The JADE MAS does not have predefined states for behaviors as with Zeus. Instead, programmers are left the task of defining functionality specific (as opposed to general states such as ready, waiting, finished) states and it is suggested that these states be implemented using a Java switch statement. Like Zeus nodes, every behavior has the option of implementing the reset method to undo state changes and therefore enable backtracking.

The JADE MAS provides a round-robin dispatching mechanism for routing incoming ACL messages. The pool of currently interested behaviors is woken-up and the first behavior has an opportunity to process the message. A logical combination of string matching on the attributes of the incoming message is performed to determine if the action method of this behavior is executed. If the matching criteria are not satisfied, the current behavior is put back to sleep and matching with the next behavior is attempted. Once a behavior accepts a message, the pool of active behaviors are put back to sleep. Explicit reposting of a message is required when it should be handled by multiple behaviors.

Behaviour is the top-level class whose children are CompositeBehaviour and SimpleBehaviour. JADE provides classes descending from SimpleBehaviour that fix the number of times the behavior runs. Classes descending from CompositeBehaviour implement support for handling multiple behaviours according to a policy. The most sophisticated class from this branch of the hierarchy is FSMBehaviour. The policy for this behavior is implemented by defining a state machine of behaviors. Because each state is itself a behavior, it is possible to embed state machines.

The FSMBehaviour class has the responsibility of maintaining the relationships (transitions) between states and selecting the next state behavior to execute. When a state is registered, a string name is associated with the corresponding behavior object. After naming all states, the transitions between them are defined and stored. Each transition is assigned an integer representing its event number. Once the start and finish states are noted, the state machine is ready for execution. When a state behavior finishes, the onEnd method returns the event number that determines the one-and-only next transition to fire. If no transitions are associated with the current event number, the default transition is taken if one was specified. Note that transitions only serve to link states; they do not encapsulate agent behavior.

3.3 Zeus

The Zeus MAS platform provides a coordination engine that executes state machines. These state machines are called graphs in Zeus. A graph defines the processing nodes of a piece of agent behavior and the associated arcs that link nodes together. The main portion of agent behavior code is divided into two node methods called exec and continue_exec. Precondition code is defined in the exec method of arcs that determine what node should be executed next. Backtracking of node traversal is supported as each node defines a reset method that is responsible for undoing processing.

The coordination engine manages executing nodes and provides convenience functions. Pseudo-parallel execution is supported by interleaving node processing using a first-in-first-out policy. Nodes typically perform some processing, are placed back on a queue while waiting for dependent processing to finish, and then re-queued for execution. There is built in support for node timeouts. The engine, to reduce the tedium of managing ACL message exchange and subsequent data structure translation, provides conversation utilities. It also has built-in methods that execute graphs for buying, selling, and a general problem solver called achieve.

All ACL messages are routed by a message handling class. This class allows rules to be registered that match incoming messages with the correct processing object. Rules are regular expression strings for specified fields of the arriving performative. For example, it is possible for a single processing object to handle all incoming messages that have a content field starting with 'R'. There is no direct support for logically connecting regular expressions. Once a rule is satisfied, the registered method of the processing object is executed with the performative as its parameter. After execution, the message handler checks the next rule to see if it is satisfied by the incoming message. This continues sequentially until all active rules are given an opportunity to process the message. Rules can be registered with the message handler dynamically. There is built-in support for one-shot rules such that after processing a message for the first time, the rule is deregistered.

Zeus provides several useful miscellaneous features. The Zeus coordination engine can process multiple graphs in parallel. Nodes are in one of several predefined states that dictate how they are processed. A visualization tool is provided that represent state information about the nodes and executing graphs. The message handler object contains statically defined rules that ensure agents follow society conventions. This is problematic when building agent societies with different norms.

4. BEHAVIOR COMPARISON

To compare MAS platform support for programming agent behavior, the meeting scheduling protocol, defined in Section 2, was implemented. A high-level parent task was defined in FIPAOS to coordinate child tasks for each of the individual states in the protocol. Return values from the children dictated entry into the next state. In JADE, a single behavior with a case statement was used to implement the meeting scheduling protocol. The Zeus implementation defined graphs for each role of the protocol and implemented the individual states using nodes. Table 2 provides a summary comparison of FIPAOS, JADE, and Zeus. In this table, PPE is an abbreviation for primitive processing element. The subsections that follow discuss these features in detail.

4.1 Primitive Processing Elements

FIPAOS includes a base task that can be subclassed but includes few other tasks that can be used as is by customized agents (tasks are included for the infrastructure agents). Each task has a task state object that keeps track of the task status. Code in the task is split primarily according to the type of messages that are handled. Aside from that imposed structure, programmers have the freedom of architecting solutions of any form. Zeus splits nodes up into two sections; one executed the first time the node is processed, and another that is executed for all subsequent processing of that node. This paradigm was useful when implementing meeting availability collection. The first responsibility of the node was to send a request and thereafter collect subsequent inform messages. Zeus nodes are always in one of several predefined states but a textual status flag can be used to communicate application specific status. Zeus provides three agent behavior graphs and their accompanying nodes and arcs. It also includes a few small protocol graphs. Similarly, JADE provides implementations for the FIPA protocols. It also includes a behavior hierarchy that was utilized in writing the meeting scheduling protocols. Unfortunately, JADE behaviors do not include predefined states and no built-in mechanism is available to note status information. All MAS platforms provide a timeout mechanism for primitive processing elements; however, Zeus does not easily support nodes with no timeout.

Table 2. MAS framework comparison

Concept Description	FIPAOS	JADE	Zeus
Support for functionally based extension of agent capabilities	task manager	agent	co-ordination engine
PPE or unit of work for a function based subsystem	task	behavior	node
PPE's provided	no	yes (individual, hierarchy)	yes (individual)
Predefined states for PPE	yes	no	yes
Aggregate container of PPE	parent task	composite behavior	graph
PPE execution scheduling	configurable round robin by default	pseudo round robin	round robin
Scheduling locus of control	centralized	distributed	centralized
Built-in support for message routing	conversation id and performaitve type	logical combination of performative attribute matching	performative attribute matching with regular expressions
Timeout support	yes	yes	yes
Protocol checking	yes	no	no
Finite state machine support	no	yes (simple)	yes (comprehensive)
Built-in data sharing across PPE's	parent-child object sharing and return value	none	input output transfer between nodes and agent-wide shared resource space
PPE code generation tool	yes	no	no
PPE execution monitoring	yes	no	yes

FIPAOS and Zeus, though suggesting nodes and tasks are reusable, fail to make a convincing argument that such is the case. In programming the meeting scheduling protocol, task and nodes tended to encapsulate minimal functionality; code was distributed across many objects. The architecture of JADE seems to encourage bigger behaviors. It may be suggested, however, that while nodes, tasks, and behaviors are the primitive processing elements; they are not reusable components because the level of abstraction is too low. Instead, composites containing these elements are more likely reusable assets that can be used to program agent behavior. These composites could be parameterized to ensure a reasonable amount of code flexibility.

4.2 Data Sharing

Support for sharing data between primitive processing elements is implemented different for each MAS framework. In writing the meeting scheduler with FIPAOS, the context object provided by the framework was used to save ACL messages and transfer them from child to parent tasks. This was needed because the states for each role in the protocol were written as child tasks spawned by a

single parent task. The parent task also used the return object of child tasks to determine state transitions. Because the JADE implementation used only a single behavior, no data sharing was needed. This framework, however, does not have built-in support for data exchange across behaviors. One solution is for parent behaviors to pass its child a reference to a common object when it is constructed. Zeus provides the most comprehensive data sharing solution as adjacent nodes pass an Object via input and output variables. The API does not support this feature, it is implicitly understood that the input and output attributes will be appropriately set by the programmer and the co-ordination engine will handle the forwarding. Zeus also has a resource database that is accessible to all parts of the agent. This is particularly useful when nodes are not related but their functionality may still affect one another.

4.3 Message Routing

Message routing in JADE and Zeus is similar while FIPAOS more rigidly defines how messages are routed to their correct primitive processing element. JADE and Zeus both use ACL message templates to match messages. JADE uses a logical combination of ACL message attributes as shown in the following code to collect meeting availability:

```
Request = agent.blockingReceive(MessageTemplate.or(
MessageTemplate.matchPerformative(ACLMessage.INFORM),
 MessageTemplate.matchPerformative(ACLMessage.REFUSE)));
```

JADE also allows string matching of ACL message attributes. Zeus uses rules written as attribute value pairs of an ACL message where regular expression matching can be specified. The GNU regular expression classes are used but the framework does not provide higher-level API calls. This forces the programmer to write cryptic matching strings with escape character sequences.

FIPAOS does not have built-in support for customizing message routing. Instead, every message is associated with a specific conversation and task. FIPAOS uses performative type to choose what method is invoked within a task. Task code is split into methods such as handlerequest or handleinform. This is fairly inflexible and depending on the application, it might not be convenient or even helpful. For example, consider a protocol that handles only requests with a differing content field.

4.4 Execution Scheduling

Execution scheduling affects agent behavior when multiple primitive processing elements are active. Though this was not the case when executing the meeting scheduling protocol, time was taken to study the source code implementations for each MAS framework. All three frameworks interleave processing elements using a round robin scheduling scheme. JADE behaviors are responsible for putting messages back on the queue if they should be serviced multiple times. This contrasts Zeus where all nodes (and customized message handlers) are given an opportunity to process a message. The architecture is slightly different in FIPAOS as task events are associated with a single task before the task is scheduled. FIPAOS and Zeus use centralized scheduling schemes. In Zeus the scheme is fixed as round robin. FIPAOS provides a default class that implements the round robin scheme. This class, however, can be customized to implement any policy.

4.5 Protocol Usage

FIPAOS comes closest to supporting protocols, the legal ordering of message exchanges between agents involved in a conversation. A FIPAOS protocol defines allowable message sequences that can occur during a conversation but does not provide an implementation for handling those messages. This is the responsibility of corresponding tasks. A meeting scheduling protocol was defined and the task manager used this to check incoming and outgoing messages for compliance. Protocols in JADE are behaviors with method stubs to handle the types of messages that are expected. The accompanying FIPA protocols provided by JADE were not applicable to building the meeting scheduling software. In Zeus, protocols are just graphs that are stored in the protocol database. This database can be accessed to dynamically load agent behavior. Typically, a higher-level graph provides a fixed agent behavior solution whose flexibility points load protocols at runtime. Though multiple protocols can be stored, the version of Zeus used did not permit choosing from the available pool. The meeting scheduling code was split into three Zeus protocols, one for each role of the conversation. Each agent loaded this protocol from the database.

4.6 Conversation Management

Conversation management in JADE is minimally supported. Tools are available to monitor and debug agent conversations but no additional mechanisms are available when programming agent behavior other than simple message routing. FIPAOS provides conversation management by checking for protocol compliance, offering message retrieval and storage utilities, conversation tracking by id, a generation tool that integrates conversation and task control code, and a messaging agent for manually driving and debugging conversations. These facilities were useful in constructing the meeting scheduling protocol although the message routing mechanism was restrictive. Conversation management support provided by the Zeus co-ordination engine was not utilized when writing protocols for the meeting scheduler. Convenience functions for managing messages were inadequate. The message retrieval mechanism translated incoming ACL messages into an internal object but some information in the original message was filtered out in the process (for example, envelope information). Monitoring of message exchanges at the society and agent level was well supported. A customized tool was developed to view messages from the entire society that was used to debug the meeting scheduling protocol.

4.7 State Machine Support

Built-in state machine support is not provided in FIPAOS. In implementing the meeting scheduling protocol, a parent task was used to manage child tasks that implement individual states. Coding the application in this way was not difficult, however, more built-in state machine mechanisms, as with Zeus, were desirable. Non-rule oriented programming in Zeus forces state-oriented programming of agent behavior. The graph, node, and arc paradigm, especially when agent behavior is defined with state charts, makes coding straightforward. Zeus keeps track of the state information for graphs and nodes making monitoring easier. Arcs in Zeus allow transitions to be defined that dictate which state to process next. In FIPAOS, the programmer must provide this mechanism. The meeting scheduling protocol was implemented in JADE using a single behavior for each role and did not utilize the finite state machine behavior class. This

mechanism, evaluated during previous research, supports state based programming of more limited scope than Zeus. State machines are composed from a top-level state machine behavior containing behaviors representing states that are connected by named transitions. When one state finishes processing, a value is returned naming the next transition to take. This infrastructure is very similar to the ad hoc FIPAOS meeting scheduler.

5. CONCLUSION

Although further analysis is needed across both the domain of problems that MAS frameworks attempt to solve and current MAS framework implementations, FIPA, JADE, and Zeus offer valuable case studies that second generation systems can leverage. Each of these systems provides reusable architectural abstractions while also sharing useful implementation patterns. While incremental development is currently improving each of these frameworks individually, the agent community at large would benefit from reusing ideas across these platforms. Development of MAS2 hopes to combine the many successful technique used by FIPAOS, JADE, and Zeus into a framework where agents can be rapidly developed and deployed.

The strong points of FIPAOS are its well-placed use of Java interfaces to separate agent subsystems, translation of incoming messages and system occurrences into events for internal processing, an isolated scheduling policy for task execution, use of a conversation object to enforce protocols and hold messages, and a task generation tool for constructing tasks from protocol definitions.

Most of the features of JADE don't stand out in comparison to FIPAOS and Zeus. Its main contributions are the quality conversation monitoring tools it provides. The Sniffer agent responsible for drawing sequence diagrams of agent interaction is valuable. The concept of programming agent behavior from a heiarchy of support classes is useful. But those classes offered by JADE, because they are primitive, don't offer programmers much in terms of reusable code. The finite state machine class is also a step in the right direction, though it is lightweight.

The Zeus state machine support and monitoring tools are well developed. A simple refactoring of the coordination engine and raising the API abstraction layer would make for a really nice solution to state-oriented programming. The regular expression message matching mechanism is also reusable. An additional layer of method calls should replace cryptic regular expression string making. Dynamic introduction of protocols into higher-level functionality is a reusable architectural element.

6. ACKNOWLEDGMENTS

We give special thanks to our colleagues at Hewlett-Packard Labs for their continued support, insights, and encouragement.

7. REFERENCES

[1] Basili, V.R. et al. Experimentation in Software Engineering. IEEE Transactions on Software Engineering. July, 1986. p. 733-743

[2] Bellifemine, Fabio et al. JADE – A FIPA-Compliant Agent Framework. Proceedings of Practical Application of Intelligent Agents and Multi-Agents. April 1990. p. 97-108

[3] Booch, Grady. Object-oriented analysis and design with applications**.** Benjamin/Cummings Pub. Co. 1994

[4] Collis, J. The Zeus Technical Manual. British Telecom, BT Labs, 1999

[5] Foundation for Intelligent Physical Agents. FIPA Communicative Act Specification, PC00037E, 2000

[6] http://fipa-os.sourceforge.net/features.htm

[7] http://herzberg.ca.sandia.gov/jess/docs/

[8] Jennings, Nicholas R. An Agent-Based Approach for Building Complex Software Systems. Comm. ACM. April 2001. p. 35-41

[9] Jennings, Nicholas R. On Agent-based Software Engineering. Artificial Intelligence, vol 117, (no.2), Elsevier, March 2000. p.277-96.

[10] Odell, James et al. Extending UML for Agents. AOIS Workshop at AAAI 2000.

[11] Poslad, Stefan et al. The FIPA-OS agent platform: Open Source for Open Standards. Nortel Networks. Manchester, UK. April 2000.

[12] Shoham, Y. Agent-oriented programming. Artificial Intelligence, vol.60, (no.1), March 1993. p.51-92.

[13] www.agentbuilder.com/AgentTools