Evaluation of the Zeus MAS Framework

Steven P. Fonseca, Martin L. Griss, Reed Letsinger
Software Technology Laboratory
HP Laboratories Palo Alto

HPL-2001-154
June 20th , 2001*

E-mail: fonseca@cse.ucsc.edu, martin_griss@hp.com, reed_letsinger@hp.com

e-commerce,
multi-agent
systems, Zeus

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 [1] 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 multiple well-know open-source agent frameworks to produce a new multi-agent system (MAS) framework called MAS2. The first step in building MAS2 is the evaluation of several agent frameworks. The focus of this paper is to collect reusable abstractions from the Zeus MAS that support plug-and-play agent infrastructure and behavior, agent interoperability, building a generic MAS core, and a MAS interface allowing domain specific extensions. The Zeus MAS framework was critiqued by implementing an e-commerce agent society.

* Internal Accession Date Only Approved for External Publication
To be published and presented in the Second International Workshop in Software Agents and Workflows for Systems Interoperability

Evaluation of the Zeus MAS Framework

Steven P Fonseca
University of California, Santa Cruz
fonseca@cse.ucsc.edu

Martin L. Griss
Hewlett-Packard Labs
martin_griss@hp.com

Reed Letsinger
Hewlett-Packard Labs
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[1] 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 multiple well-known open-source agent frameworks to produce a new multi-agent system (MAS) framework called MAS2. The first step in building MAS2 is the evaluation of several agent frameworks. The focus of this paper is to collect reusable abstractions from the Zeus MAS that support plug-and-play agent infrastructure and behavior, agent interoperability, building a generic MAS core, and a MAS interface allowing domain specific extensions. The Zeus MAS framework was critiqued by implementing an e-commerce agent society.

1. Introduction

In evaluating the Zeus MAS, it is first necessary to identify the development issues of building an object-oriented framework, define a compelling problem that will produce insight into the agent domain, and introduce the core capabilities of the Zeus MAS. Included in this introduction are brief explanations for Zeus classes that are referred to in the subsequent sections entitled agent behavior, agent infrastructure, MAS platform infrastructure, and the FIPA standard.

1.1. Agent frameworks

A MAS written in Java is just an object-oriented application framework for the agent domain. The challenges of developing and programming with a multi-agent system are identical to those for building and using an object-oriented application framework. The most significant factor effecting the development of agent systems today is not understanding the agent domain and subsequently not having reusable abstractions to guide development. Other challenges associated with frameworks are the steep learning curve faced by application programmers, lack of explicit control flow, difficulty removing code defects, and integration with other frameworks, legacy systems, and other components [2]. Understanding these challenges helps establish the context of evaluating multi-agent systems and developing MAS2. Some of the very issues listed above were encountered while programming with Zeus. The recurring theme when building the e-commerce agent society was that a successful MAS must strictly adhere to object-oriented and framework design principles while also employing current state-of-the-art practices from the artificial intelligence community.

During the evolution of a framework, multiple design and implementation iterations are required to refine domain knowledge and make corresponding changes to the framework [2].

1.2. An e-commerce scenario

Serving as motivation for choosing the scenario to evaluate Zeus is the belief that software agents and mobile appliances are technologies that have the potential to change the way people purchase products by connecting the physical presence of stores with their Internet representation and delegating consumer tasks to intelligent pieces of autonomous software. The archetypical example is the shopping mall of the future; shoppers use personal digital assistants (PDA) to view web pages (while at the store) that extend store services, mall-wide services are also available through the PDA, and intelligent agents negotiate for desired products based on shopper preferences. A lightweight version of this scenario was implemented, however, the mobile appliance technology and infrastructure elements of the scenario are not discussed herein. Only the negotiation and mall facilities portions of the e-commerce scenario are shared because Zeus was evaluated by focusing on this code and its API.

In the e-commerce scenario, agents purchase products by participating in an English auction. This protocol is composed of four roles including the auctioneer, seller, bidder, and facilitator. An agent in the e-commerce society assumes a single role. Note also that message types (inform, subscribe, etc.) passed between agents are taken from the FIPA Communicative Act Library Specification [3].

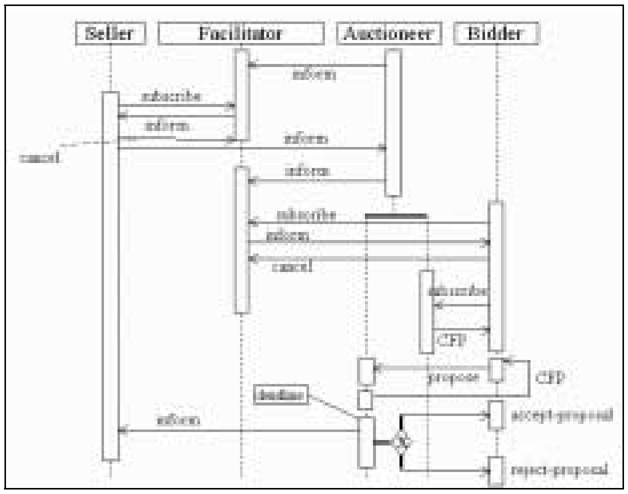

Figure 1: AUML English auction sequence diagram.

The general conversation flow beings when the auctioneer registers with the facilitator to inform the society that it can hold auctions. Then sellers can subscribe to the facilitator for a list of available auctioneers. The facilitator sends the names of available auctioneers to the seller who can then request that a product be auctioned. In response to this request, the auctioneer informs the facilitator that it is selling a product. This initiates the auction. Agents (future bidders) wishing to purchase this product consult the facilitator who informs them of auctions that are currently open. The auctioneer waits for bidder agents to register. Once registered, bids can be placed. Whenever a new high bid is received, the auctioneer informs all registered bidder agents. Bidding continues until a fixed time has passed. At the close of the auction, the auctioneer informs the agents who has won. Though the sequence diagram does not show the payment conversation, the winning bidder and seller engage in this message exchange.

1.3. Zeus MAS primer

British Telecom labs developed the Zeus MAS framework. The MAS development environment consists of an API, code generator, agent and society monitoring tools, programming documentation, and three case studies (including a sample fruit market). The platform is written in Java and is open-source. The Zeus version evaluated was 1.03b, released in the summer of 2000.

A complete Zeus agent has a coordination engine enabling functional behavior organized around conversation protocols, a planner that schedules sub-goal resolution, an engine for rule-based behavior, and databases to manage resources, abilities, relationships between agents, tasks, and protocols. This evaluation of Zeus focuses on the coordination engine, protocols and strategies, abilities (more commonly called services), and infrastructure agents. These agents include the visualizer for monitoring an agent society, the nameserver for address resolution, and a facilitator that matches service providers with service requesters. What follows is a description of the core classes for implementing agent behavior and communication.

Agents in Zeus interact via message exchanges. One mechanism for specifying agent behavior in Zeus is to associate messages having certain values to executable pieces of code. All agents participating in a Zeus society have a message handling class that is responsible for retrieving incoming messages and dynamically executing code. Rules are registered with this message handling class when it is instantiated and throughout the agent's lifetime. Message rules explicitly link the messages with the methods that should be invoked. The rules that are added when the message handling class is instantiated provide most of the behavior that is required of all agents interacting in a Zeus society.

Zeus agents pass string-based messages to communicate. When sending a message, a new Performative object is instantiated and its fields are set (including message type, sender, receiver, etc.). This object is placed in the outgoing queue of the MailBox. A PostMan thread continually retrieves messages from the queue and is responsible for their transmission. The PostMan resolves the address (querying the AddressBook) of the receiving agent and opens a socket connection using the correct host name and port number. The receiving agent's Server creates a Connection object for the incoming message. The Connection object translates the byte stream into a performative that is placed in the incoming message queue. The MsgHandler routes Performatives to their correct execution objects.

A coordination engine is provided to execute protocols. A protocol is a defined series of message exchanges and accompanying processing (behavior). Agent behavior is broken down into nodes that are executed by the engine.

2. Agent behavior

The Zeus MAS provides some support for constructing agents but falls far short of providing a comprehensive solution for composing agents from reusable behaviors. Suggestions for improving the modularity of Zeus are provided and the limitations of not having meta-negotiation are discussed.

2.1. Societal behavior

It is valuable for agents built from multi-agent system platforms to recognize required society messages and have the ability to process them. Zeus provides this agent behavior. While helping promote reuse, however, to fully leverage code it must also be adaptable. The problem with the Zeus architecture is that the agent behavior (registration with the Visualizer, etc.) resides almost completely in the message handling class. Anytime a new agent is created, the same message handling class is instantiated. While this ensures that all agents in the society are provided with the same society required capabilities, it does not give programmers the ability to easily redefine societal behavior.

Practicing object-orientation requires that objects have clear and intuitive functionality. Objects are composed of cohesive attributes and methods that provide a focused and related set of capabilities [4]. Based on these principles, the message handling class should not be responsible for containing methods to process the required Zeus societal messages. Instead, it can be argued that the responsibilities of the message handler are to receive incoming messages and forward them to appropriate helper objects for further processing. This architecture provides a more clean and clear interface between receiving a message and subsequent processing of that message.

The flexibility gained by uncoupling message receiving from agent behavior makes the Zeus multi-agent platform more flexible and extendable. This allows multi-agent society developers to specify the rules or conventions that agents must follow. Previously, a Zeus agent society was defined by the behavior in the message handler. Now one can view the Zeus agent society as all the classes that are reused to support agent interaction. These classes enable agent interaction but do not dictate how that interaction takes place.

At the agent level, separation of message receiving from behavior enables agents to dynamically switch the societal conventions they follow. While the benefits of this are not apparent in considering a single society, consider the electronic commerce domain where businesses are likely to have their own interaction conventions. Agents interacting with multiple businesses may need the ability to switch the standard set of rules that define their behavior. Similarly, in the case of mobile agents, societal conventions may depend on the current host computer.

Society classes can be used to specify and provide the fundamental abilities agents need to posses. The Zeus platform could contain classes that provide implementations for standard models of interaction. In this scenario, a catalog of multi-agent society patterns and their implementation classes are provided for Zeus developers. It can be imagined that these society classes could be adapted using inheritance or by configuring society objects using an API. Through the appropriate use of Java interfaces, the Zeus platform could also support customized society objects developed by users when the built-in society objects are unsatisfactory. Zeus developers could then share these classes in much the same way as classes (API's) are shared in the Java community.

In addition to improving the flexibility of the Zeus platform, encapsulating message types and corresponding processing methods into society classes centralizes the code that partially defines agent behavior. Developers know where to search for society level functionality. In the current Zeus platform it is not intuitive to look in the message handling class for society level functionality. Further, the MailBox and Engine also contain society required agent behavior and thus force developers to search multiple classes. Searching multiple classes is not prohibitive when all classes are conceptually well connected to each other and to the behavior they provide. This principle, however, is violated by the MailBox object because when instantiated it registers its agent with the nameserver.

Centralization of society functionality using a single class or cluster of cohesive classes also makes documentation and maintenance easier.

2.2. Implicit protocol agreement

In both the FruitMarket case study and our implementation of an English auction, the negotiation protocol is specified at compile time. Compatible protocols are loaded into the participating agents protocol database and are used as the only coordinated communication mechanism for interaction in the society. A single communication method is sufficient and favorable when demonstrating the functionality of a protocol. A single protocol is too restrictive for marketplaces composed of heterogeneous merchants and consumers. It is anticipated that marketplace participants will engage in conversations with different conventions. This requires the ability to understand multiple protocols, agree on a negotiation protocol, and switch between protocols at runtime.

The Zeus MAS partially supports the use of multiple protocols. It is possible to store multiple protocols in an agent's protocol database, however, it is not possible to easily select the protocol that should be active during a negotiation. For example, there is no selection process for loading the negotiation protocol that is used when trying to sell an item. Zeus is hard-coded to retrieve the first protocol stored in the protocol database. Furthermore, there is no built-in support for meta-negotiation. A standardized way for agents to agree on the protocols to use during the negotiation process is required for all but the simplest of agent societies.

2.3. Protocol and strategy parameterization

While implementing an English auction, it was recognized that several variations of this common protocol could be created with minimal code changes. Rather than create a suite of nearly identical protocols, a better solution is to pass a configuration object that specifies the flexibility points and therefore characteristics of the auction. Some constraints that could be configured include the minimum bid, minimum bid increment, and the elapsed time required before a bidder is selected as the winner. One could also imagine other variations of an English auction that are not sufficiently different to warrant their own name.

2.4. Agent and domain API's

Two primary problems were encountered while programming with the Zeus API. First, no interfaces were defined for the subsystems for which an agent is composed. Second, The API provided methods at an abstraction level that was too low.

While high granularity code promotes more flexible programming, as was found out using Zeus, it can also overly burden programmers by requiring them to manage too many details. A multi-level API is a possible solution. The low level API could be composed of classes that, when used in combination would form a second level API supporting programming concepts from the agent domain such as achieving goals, sending messages, or changing state. The domain level API would be at the highest level. In the case of e-commerce, concepts such as buying and selling would be directly supported. Unfortunately, the Zeus API is composed mostly of level-one concepts, augmented with a couple of e-commerce domain concepts. If done correctly, these domain concepts are supported by lower-level API calls. This is not the case in Zeus because concepts such as buy and sell are not built from general behavior methods from a lower abstraction level.

Zeus agents are composed of subsystems that are tied together using a container (AgentContext) object. By programming convention, a reference to the container object is generally available. The AgentContext provides basic access methods for retrieving references to the subsystem objects (planner, rule engine, etc). Passing subsystem references results in poor encapsulation. Although building an agent from components that intuitively and logically split functionality is the first step toward reusable agent behavior, interfaces to these subsystems must exist and explicit links to the objects must also be removed. Interfaces establish pseudo standards making it possible for the agent to be composed of components from multiple vendors. Zeus provides implementations for its subsystems and provides no mechanism for leveraging work by other developers in the agent community. A notable example of composing an agent from multiple developers is the JADE and FIPAOS [5,6]platforms. They both provide the means to use JESS, a third-party open-source rule engine.

3. Agent infrastructure

The topic of agent infrastructure is meant to include the agents provided by Zeus to facilitate interaction in the society and also the core components from which all Zeus agents are built.

3.1. Behavior engine

The coordination engine is a useful abstraction that could benefit from refactoring. The engine paradigm successfully separates agent specific functionality into behavior elements (nodes), executes behavior using an interleaving scheduler, supports time limited behavior, and provides event monitoring at multiple levels of abstraction. A weakness of the engine is the lack of generality of its methods. Also causing difficulties while implementing the English auction was engine-based message retrieval.

For new and continuing dialogues, the engine provides methods for sending and receiving messages to alleviate programmers from writing tedious and redundant code. The problem is that incoming messages are converted into an internal data structure. Information contained in the message is lost during this translation. For example, the replyWith and envelope fields of a message are not transferred when using the continue_dialogue method provided by the Zeus coordination engine. A central message storing mechanism is a useful abstraction, however, this facility should only store messages and forward copies to their owner when requested. Subsequent processing of the message remains the responsibility of the message owner.

Buy, sell, and achieve methods are provided by the Zeus coordination engine. The code for these methods is essentially the same, the only difference being the type of graph that is run. The graph type is hard-coded into the method when it should be passed as a parameter. This lack of generality required a Zeus source code change to implement the English auction. Buy and sell are concepts from the e-commerce domain and not the agent domain. While it makes sense to support domain specific extensions to a MAS framework, putting this functionality in the core MAS classes is inappropriate.

3.2. Facilitation

Zeus provides a facilitator agent to serve as a yellow pages service for other agents in the society. In Zeus, agent services are called abilities. These abilities are described using concepts from the active ontology. The ontology is used by the facilitator to match the attributes of an ability with requests for that ability. Unfortunately, the matching service is not sophisticated enough to handle the IS-A relationship. For example, if a Macintosh (as in apples) is advertised for sale, the facilitator will not inform agents wanting to purchase any type of apples of Macintosh availability. Assuming Macintosh is a sub-concept of apple, one would intuitively expect a match.

3.3. Content parsing

Zeus provides all agents with a parser for translating the content field of messages into objects. This parser has decoding methods to convert the string content field into a corresponding object. The problem is that the type of object that is sent in the content field must be known prior to receiving a message so the corresponding method can be used for parsing. Because the facilitator and nameserver were inadequately documented, decoding messages from them was difficult because it was not easy to match the string representation of an object to itself. Furthermore, strings were appended to the beginning of the content field by the nameserver and facilitator to further specify the message meaning. This ad hoc content language is too cryptic.

The poor parsing mechanism in Zeus could be replaced by using XML as the content language. Given the document type definition, agents could parse content information using general code as opposed to using message specific methods. The string version of the content field would be readable. Further, since XML is a popular technology for storing information, programmers are not burdened with learning yet another language.

3.4. Message transport encapsulation

A flexibility point of possible importance to multi-agent system platforms is the transport layer for agent communication. This layer is responsible for transmitting raw data across the network.

Multi-agent system developers can choose between at least two transport options when building an extensible communication subsystem. The MAS can be built such that one transport mechanism (HTTP, for example) can be

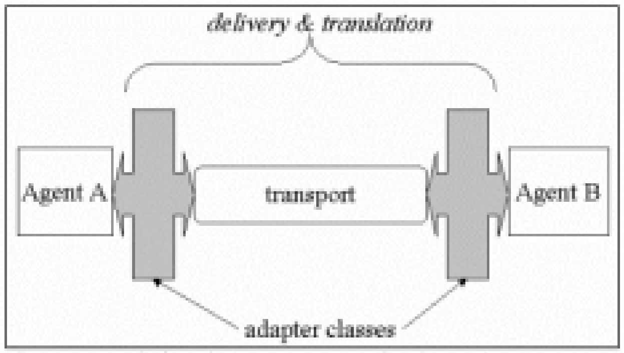

Figure 2: Isolating the transport mechanism.

easily compiled with the message subsystem. Alternatively, the MAS could support runtime selection of a message transport mechanism (possibly on a message by message basis). In either case, the interfaces between the transport layer and the MAS communication subsystem must be specified. Minimally, this requires an interface for sending and an interface for receiving messages. It is advantageous to use adapter classes at these interfaces to keep classes implementing the transport layer separate from those classes comprising the MAS communication infrastructure. This is shown in Figure 2. Consider the transport mechanism and corresponding adapter classes as a single software component. A MAS could support plug-and-play component reuse for message delivery and translation simply by establishing a fixed interface that component developers could connect to using adapter classes. The underlying architecture key is requiring that no dependencies (coupling) exist between MAS classes and the transport classes.

Given the need for an interface between the MAS communication subsystem and the transport mechanism, interface design must be addressed. First, adapters for sending messages must translate the recipient name into their address. Addresses could take the form of a host and port number (as with TCP/IP Socket) or a URL for HTTP-based communication. Second, observe that a single versus multi-transport mechanism has a simpler interface. With multi-transport systems, agents must provide the transport mechanism in addition to the message itself when sending a message. Third, the adapter should be able to forward incoming messages to other servicing objects. Figure 3 shows an EDI adapter that receives an incoming EDI transmission. The adapter has the ability to pass the data to an EDI handler object or translate the data into an ACL performative and forward it to the central ACL performative message handler.

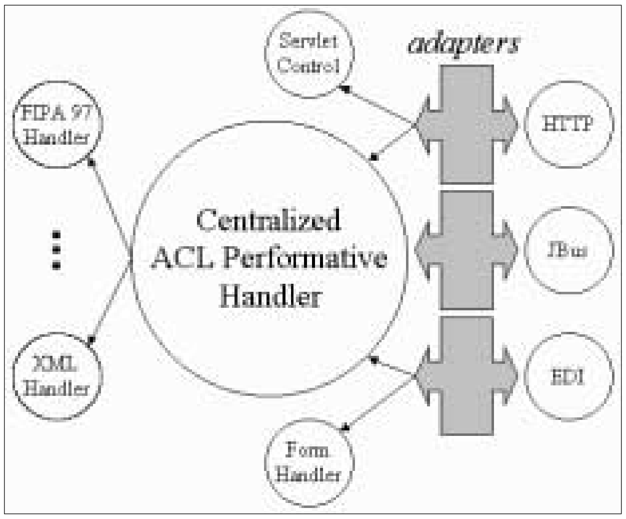

Figure 3: Multi-transport coordination.

Though the Zeus Technical Manual states that it is "possible to replace the TCP/IP mechanism with a middleware alternative" [7], it is not a simple endeavor because interfaces were not defined and there is prohibitive coupling between classes. A Performative object is the informal interface to the transport mechanism. But good design would establish Java interfaces on the sending and receiving ends of the transport mechanism to isolate it from the rest of the agent. Further, an agent does not have the ability to store addresses that do not follow the host/port convention. Therefore, coupling exists between the message transport and address book of the agent.

3.5. Multi-transport support

One can imagine that alternate multi-agent system deployments would have different transport requirements. Consider several examples: 1) In the e-commerce domain it may be desirable to securely transport some messages. 2) HTTP based message exchange might make sense when Internet-based agent control is desirable. 3) The message exchange patterns in a multi-agent system could dictate the appropriateness of transport protocol selection. If two agents regularly engage in conversation, then it is more efficient to use a transport layer with dedicated connections to avoid the setup performance penalty associated with establishing a connection on a message-by-message basis. Additionally, a publish-subscribe transport such as Java JMS could be more appropriate when conversations require regularly broadcasting messages to the agent society. 4) The recipient of a message could dictate the required transport and may not be another software agent.

It is conceivable that a given domain might benefit from having multiple transport mechanisms for more efficient communication. An agent could rely on its conversation manager to switch transport mechanisms based on the past message exchanges.

4. MAS platform infrastructure

To successfully develop agent societies, programmers must minimally be supported with good documentation and have the ability to view agent state and interaction. A third component that a MAS platform infrastructure might provide, as Zeus does, is a code generation tool to build agents.

4.1. Monitoring and managing

The Zeus MAS provides GUI-based views for monitoring the society as a whole and also for individual agents. An infrastructure agent called the visualizer controls the society wide viewer. It provides a number of different graphical representations of interaction including such things as the number of messages sent between agents, the type of messages sent by each agent, animated display of message exchanges, goal resolution, and strategy graphs. The majority of the views the visualizer provides are visually appealing but lack utility. Conversation management facilities are completely lacking but should be included with any comprehensive MAS development environment. Of use to programmers are conversation sequence diagram generation, a tool for constructing conversations, conversation recognition and verification, a view of message traffic using filters to organize their presentation, identification of conversation roles, ontology usage, agent state, society state, and tracking conversation context.

While Zeus fails to provide a compelling solution to society level monitoring and management, it provides a very well done agent-monitoring interface. All aspects of agent state are visually represented, incoming and outgoing messages are available, and the runtime state of conversations is pictorially represented as a color-coded graph. The elements of this GUI serve as an excellent model for viewing agent state and behavior.

4.2. Code generation

While the code generation tool provided by Zeus was not fully evaluated, for configuring a simple agent with strategies, protocols, and resources it was more burdensome than beneficial. For example, the interface is difficult to use, programmers are required to know in advance the strategy parameters and legal values, and adding to the list of known protocols is cumbersome. To sidestep these problems, XML agent configuration files were used and a parser was developed to write source code for initializing agents. Intuition suggests that the code generation tool may prove more useful when writing agents that utilize the Zeus rule engine. As with other portions of Zeus, the code generation tool is conceptually a good idea, however, the current implementation needs improvement.

4.3. Documentation

Documentation is critical to efficient application framework programming. It well known that learning a framework, regardless of domain, is costly and time consuming [8]. Work on the e-commerce scenario was inhibited because the Zeus API and source code are poorly documented. Imagine trying to program with an API that provides little or no information about classes, methods, or attributes. The only recourse was studying sparsely commented source code, which defeated the purpose of having an API.

Case studies, an application guide, technical manual, and role-modeling guide supplemented the API documentation. While this documentation did offer insights into the Zeus MAS, topics were never covered fully enough to enable solving significant programming problems. It was always the case that additional information, elicited from the source code, was always required.

5. FIPA standard

At the time Zeus was written, the Foundation for Intelligent Physical Agents (FIPA) was beginning to develop specifications for multi-agent system development, agent-to-agent communication, and domain specific application of agents. The Zeus communication subsystem was written to conform to what are presently the ACL Message Structure and Communicate Act Library specifications. The infrastructure agents provided by Zeus speak the FIPA agent communication language (ACL) and parsing classes are available to decode messages following the FIPA performative syntax.

After version 1.03b of Zeus was released, FIPA wrote an Agent Management Specification that, among other things, defines the legal messages for interfacing with a FIPA compliant name service and directory facilitator. The advantage of this, and what was lacking when developing the English auction, was a standardized and documented way of interacting with the society support agents. For example, writing a routine to communicate with the facilitator required looking at its message processing code to determine the message format it expected for service queries. The disadvantage of following the FIPA standard are the message inefficiencies paid for not tailoring message interfaces between agents to the domain. The amount of information required in a FIPA compliant message does not seem appropriate for simple domains or simple solutions.

Another concern of FIPA is the process used to generate its specifications. It is generally understood that frameworks evolve by applying them to solve problems, evaluating their weaknesses, adjusting domain models, and then refactoring the implementation [9]. Solving "real" problems of importance drives this process. What's troubling is that FIPA is generating specifications based on conceptual design and not from direct experience. These specifications are used to guide MAS framework implementations. It seems more appropriate to define specifications based upon the collective experience of MAS framework developers as they attempt to solve problems from their domains of interest.

6. Conclusion

Construction of next generation MAS frameworks can benefit from the first round of abstractions elicited from the multi-agent system domain. British Telecom's main contributions are the identification of valuable agent concepts and component design for executing agent behavior using a protocol-based paradigm. Most of the Zeus MAS framework requires refactoring. The lesson is that multi-agent system design must follow object-oriented framework design principles if a development environment that offers significant design and code reuse is desired.

Further domain analysis is needed across both the domain of problems that MAS frameworks attempt to solve and the current MAS framework solutions.

The Zeus high-level architecture must be replaced with a flexible alternative that enables agents to be composed of subsystems from potentially different developers. Establishing interfaces among subsystems is a possible solution.

The success of a multi-agent system platform depends on the same factors that make any framework successful. At a minimum, a MAS should provide adequate documentation, a usable API, monitor and debugging tools, capture the essential concepts of the domain, and support points of variability. The struggle to achieve these design criteria will continue until the multi-agent system domain is well understood. Until such time, the iterative and incremental process of refining the domain model and architecture continues. Successful MAS implementations will follow.

7. Acknowledgement

Special thanks to team members David Bell, Harry Chen, Ye Chen, Dick Cowan, Peter Finin, Ed Katz, Thomas Raffill, and Farrell Wymore.

8. References

[1] www.agentbuilder.com/AgentTools

[2] M. Fayad, D. Schmidt, R. Johnson, *Building Application Frameworks: Object-Oriented Foundations of Framework Design*, John Wiley & Sons, New York, 1999

[3] Foundation for Intelligent Physical Agents, “FIPA Communicative Act Specification”, PC00037E, 2000
[4] G. Booch, *Object-Oriented Analysis and Design*, Addison-Wesley, Menlo Park, 1994
[5] F. Bellifemine, G. Caire, T. Trucco, G. Rimassa, “Jade Programmer’s Guide”, CSELT, 2000
[6] Nortel Networks, “FIPA-OS V1.3.2 Distribution Notes”, Ontario, 2000
[7] J. Collis, “The Zeus Technical Manual”, British Telecommunications, BT Labs, 1999
[8] S. Fonseca, “Object-Oriented Application Framework Documentation”, Master’s thesis, UC Santa Cruz, 2000
[9] K. Czarnecki, U.W. Eisenecker, *Generative Programming*, Addison-Wesley, Canada, 2000