An Internal Agent Architecture for Dynamic Composition of Reusable Agent Subsystems – Part I: Problem Analysis and Decomposition Framework

Steven P. Fonseca
Software Technology Laboratory
HP Laboratories Palo Alto
HPL-2002-193
July 10th , 2002*

E-mail: fonseca@cse.ucse.edu

agent architecture, MAS framework, FIPAOS, JADE, Zeus

The internal agent architectures that current MAS frameworks provide don't enable the rapid implementation of agents from reusable components. This is because of the underlying problem that the agent-oriented programming paradigm, defining how abstraction, decomposition, and modularity are achieved, is not sufficiently understood or developed. This paper presents an agent decomposition framework that offers agent-oriented programming and software engineering separation of concern guidelines and discusses the weaknesses of agent architectures currently supported by MAS frameworks. This is in preparation for proposing a more flexible and extensible internal agent architecture than current MAS frameworks provide. It is envisioned that this architecture will provide an infrastructure that supports constructing agents from reusable components at the subsystem level. Subsystems are added to an agent through a dynamic description, event, and ontology registration process. Once connected, the composite subsystems interact by an event-based software bus that acts as the central nervous system of an agent. This gives subsystems the ability to reason about the functionality and current state of their constituent parts and allows agents to be composed from industry wide best known components instead of building agents from a single MAS framework repository.

* Internal Accession Date Only
Approved for External Publication

An Internal Agent Architecture for Dynamic Composition of Reusable Agent Subsystems Part I: Problem Analysis and Decomposition Framework

Steven P Fonseca

UC Santa Cruz / HP Labs
Baskin School of Engineering
Santa Cruz, CA 95064
fonseca@cse.ucsc.edu

Abstract. The internal agent architectures that current MAS frameworks provide don't enable the rapid implementation of agents from reusable components. This is because of the underlying problem that the agent-oriented programming paradigm, defining how abstraction, decomposition, and modularity are achieved, is not sufficiently understood or developed. This paper presents an agent decomposition framework that offers agent-oriented programming and software engineering separation of concern guidelines and discusses the weaknesses of agent architectures currently supported by MAS frameworks. This is in preparation for proposing a more flexible and extensible internal agent architecture than current MAS frameworks provide. It is envisioned that this architecture will provide an infrastructure that supports constructing agents from reusable components at the subsystem level. Subsystems are added to an agent through a dynamic description, event, and ontology registration process. Once connected, the composite subsystems interact by an event-based software bus that acts as the central nervous system of an agent. This gives subsystems the ability to reason about the functionality and current state of their constituent parts and allows agents to be composed from industry wide best known components instead of building agents from a single MAS framework repository.

1 Introduction

Architecture and code reuse are two primary means of systematically building quality software systems [3]. Object-oriented software development has highlighted the importance and success of building from collections of compatible components [2]. Object-oriented frameworks offer sets of components designed to interact with each other (via patterns) to solve domain specific problems and developers also recognize the importance of, and have deployed, comprehensive multi-framework solutions [3]. The keys to this success are clearly delineated component abstraction layers, appropriate system decomposition, collections of reusable and compatible components, interface specifications, interoperability between frameworks, and suitable architec-

tures for connecting components together. These mechanisms for the agent-oriented paradigm are underdeveloped and immature. In the context of proposing an internal agent architecture for composing an agent from reusable subsystems, this paper addresses a subset of the fundamental problems confronting agent-oriented software engineering: 1) In what ways is the agent-oriented paradigm different from, or an improvement to, object orientation? 2) How can existing MAS frameworks be evolved to make agent construction easier? 3) Can an internal agent architecture be developed that facilitates better component reuse? 4) How is agent behavior decomposed into reusable components and at what granularity?

Currently, there is little consensus on how to decompose agent behavior and even the abstractions afforded by the agent-oriented paradigm have not stabilized. While agents constructed from different multi-agent system frameworks can interoperate thanks to standardization efforts such as FIPA [6], the framework components they are composed from are not, in general, designed for use with other MAS frameworks. Further, MAS framework and agent infrastructure architectures are often highly coupled with the subsystems they provide and consequently this limits flexibility and the abstraction level that component reuse and adaptation can occur. While the usefulness of composing an agent from independent yet compatible subsystems is evidenced by standard object oriented design practice, the promise of ubiquitous e-services makes an even more compelling argument for flexible and dynamic agent composition.

The agent oriented software engineering problems discussed are one of several key technical problem areas inhibiting the adoption of agents as an industry wide solution for software systems that are distributed, heterogeneous, and autonomous. The most high-level question, can agent technology can be used to build industrial strength software, cannot be answered without overcoming its engineering problems. For if a large body of programmers does not have the software infrastructure necessary to build many agent systems, there will be neither sufficient interest in, nor feedback to the design process, for agents to flourish. Research and development efforts at Hewlett Packard Labs were confronted with this reality in developing an e-commerce environment for mobile shoppers [5]. While our primary goal was to develop a testbed for studying the behavior of agents and agent societies for the electronic commerce domain, one of our main lessons was that support for designing, implementing, and deploying agents was inadequate [4, 6]. Subsequent to this experience, research time has been spent developing agent-oriented programming principles and infrastructure while working in parallel on agent-oriented software solutions.

Comprehensive MAS frameworks offer a collection of functionality encapsulated within an agent. An excellent example is agents built using the Zeus MAS framework that are composed of a message transport, behavior execution unit, rule engine, central storage mechanism, event system, and abilities database. In the case of Zeus, a highly coupled infrastructure connects these subsystems together. Virtually no interfaces are used, the event system is based on predefined static event types, subsystems must use the Zeus ontology model to leverage classes offered by the API, an internal data format must also be used, and extensibility of an agent is only possible at a few flexibility points that are at an abstraction level lower than a subsystem. The architecture of Zeus suggests that emphasis was placed on developing a comprehensive closed solution whose extensibility is offered at a single level of abstraction.

In contrast to Zeus, the internal agent infrastructures provided by the FIPAOS and JADE MAS frameworks are lightweight and do use interfaces, though not often enough, to decouple the components that compose an agent. These frameworks offer less functionality than Zeus but do provide message transport and routing, a behavior execution unit, ontological support, FIPA protocol support, a varying degree of conversation management, and life cycle support. The FIPAOS and JADE frameworks, like Zeus, support agent extensions at only one level of abstraction that is finer grain than an agent subsystem. Further, the infrastructure of these two MAS frameworks is too lightweight and consequently programmers are left coding infrastructure for their applications. Not only is this work redundant, but also organizations cannot take advantage of the design lessons these framework developers have accumulated and that should have been captured and offered as reusable components and patterns. These MAS frameworks do not provide an internal agent architecture that supports component reuse that correctly balances the tension between structure and flexibility. JADE is too flexible as it provides next to no infrastructure, while FIPAOS is also architecturally thin and what is provided is too rigid. Further, these frameworks provide only minimal infrastructure, primarily in the form ontology support, for the components of an agent to interact. A detailed discussion of the current limitations of MAS frameworks is given in Section 3.

The inadequacy of the internal architectures for FIPAOS, JADE, and Zeus agents are representative of the problems common to MAS frameworks in general. While each is an excellent first attempt to develop a collection of classes for constructing agent societies, their implementations require refactoring and their architectures improved. While active open-source communities support the evolution of these frameworks, there is an even greater opportunity for advancement by capturing the best designs and implementations of these efforts and using them to build a second-generation MAS framework. Further, an open internal agent architecture encourages collaboration across research groups thereby fostering an increase in component reuse and standardization.

The internal agent architecture proposed solves many of the common problems of existing MAS frameworks. Its design requirements include an infrastructure for the dynamic composition of an agent from subsystems, an event system that allows dynamic registration of event types and whose events can be shared at the subsystem level, a subsystem description mechanism, a common internal architecture ontology, dynamic registration of subsystem ontologies, an internal data marshalling mechanism, and an execution scheduler. These requirements are based on two years of programming with FIPAOS, JADE, and Zeus frameworks developing applications in the meeting scheduling and electronic commerce domains at Hewlett-Packard Labs and UC Santa Cruz.

The next section provides a framework for decomposing agent behavior by explicitly describing the agent-oriented programming abstraction layers and the types of components used to modularize and decompose multi-agent systems. Particular emphasis is placed on describing the programming of internal agent components as it is the responsibility of an agent architecture to facilitate their interrelationships. Following this foundation, Section 3 evaluates three open-source Java MAS framework architectures

2 Agent Abstractions and Decomposition

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 must be used effectively [2, 7]. In a very general sense, the purpose of this 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 [7]: "What are the essential concepts and notions of agent-based computing."

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 request agent
Agent Role	A well-defined high-level agent action	Meeting participant
Agent Work Unit	Primitive element encapsulating agent functionality	FIPAOS task, JADE behavior, Zeus graph and nodes
Application Framework Extensions	Application specific subsystems not part of the agent framework core	JESS
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, JFC
Language	Core software foundation	Java

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 layers of abstraction proposed in this paper and corresponding decomposition framework appear in Figure 1. 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 MAS frameworks recast object calls using agent-oriented concepts. The subsections that follow cover the MAS framework, framework extensions, role, and agent levels of abstraction in detail because it is at these levels that reuse currently occurs.

2.1 MAS Frameworks

An object-oriented framework provides a base of code for a specific application domain, that adheres to an architecture, establishes the relationship between classes through inheritance and references, and is extensible. MAS frameworks are a specialty of object-oriented frameworks with an application domain of distributed and autonomous software using agents as the main unit of encapsulation. It follows that a

successful MAS framework should adhere to well-known object-oriented framework design and implementation principles.

It also follows that a MAS API should encapsulate lower-level concepts with next level agent-oriented abstractions. The question is what set of agent-oriented abstractions should be provided by a general purpose MAS framework? How is this functionality split into reusable modules? And how can the framework be easily extended? These questions have not been definitively answered but one can look to the FIPA and JAS specifications that are on the path to extracting the essence of multi-agent system software for a partial answer.

The main contributions of the FIPA specifications are a communications and agent services infrastructure for multi-agent systems. The intuitive assumption of FIPA is that all multi-agent systems have a core set of functionality that varies little from implementation to implementation and agent communication must be standardized to facilitate interoperability. FIPA abstractly defines a set of requirements with the goal of providing general multi-agent system specifications that MAS framework developers can implement. The JAS specification serves as an intermediary between FIPA and framework developers by providing an unimplemented reference API. This API proposes the classes, methods, and interfaces used to construct communication messaging, name resolution, and directory facilitation. FIPA and JAS have provided the set of agent-oriented abstractions that should be provided by a general purpose MAS framework for these three core capabilities.

FIPA defines several other specifications addressing, for example, the agent lifecycle and communication protocols. Peripheral specifications should be delineated from those specifications that enable interoperability to clearly communicate what FIPA compliance guarantees. A number of these specifications address issues whose general applicability is questionable. Should FIPA be responsible for establishing e-commerce protocols? Application domain specific specifications should be left to their respective experts. Appropriate separation of concerns is vital to component abstraction, modularity, and reuse. MAS framework design can be decomposed into the core capabilities required for all systems and an application specific portion. MAS framework design could integrate the two, as is current practice; more flexibly, a generic MAS framework could provide facilities making it easy to add large domain-specific extensions at runtime or when applications are developed. This is the essential goal of developing an internal agent architecture that supports subsystem composition.

An equally important input to the development of agent-oriented programming abstractions and corresponding framework API's are the plethora of first generation MAS frameworks [1] available today. In studying these frameworks, it is apparent that the software core of a general MAS framework includes more than just the communications mechanisms described by FIPA and JAS. For example, many of these frameworks also include agent shells and a behavior extension mechanism for adding application specific code. One can imagine standardizing a state-based behavior execution unit. This is analogous to the JRS (Java Rule Engine Specification), although the JRS is not agent-specific. Additionally, conversation management facilities are frequently provided. FIPA and JAS do not offer any guidelines for how to route incoming messages, maintain conversations, or monitor agent interaction.

Current MAS frameworks, in addition to illustrating the common elements of a MAS framework, also offer insight into the ways that these systems vary. The more than 60 MAS frameworks [1] available offer a wide-range of functionality providing their own unique combination of reusable components that support specific application features (i.e. mobility) desired by the framework developers; domain dependent agent requirements often dictate framework design choices. It is these variability points that must be abstracted and systematically encapsulated into modules whose interface with a generic MAS framework core is clean and well defined. Restating this idea, a MAS framework should be composed of subsystems that are well encapsulated, systematically connected with other subsystems, and that can be used to customize an agent shell from which additional behavior extensions are built.

2.2 MAS Framework Extensions

If a generic MAS framework only provides the core capabilities required of an agent, such a system must be conveniently extendable. The internal agent architecture proposed in a future paper enables component reuse at the MAS framework extension level of abstraction by constructing agents from pluggable subsystems. These subsystems might be general-purpose engines that can be further specialized. They could also encapsulate application specific code offering a complete set of functionality for an agent to utilize. Some combination of these two extremes is also possible. For example, a MAS framework could be supplemented with a generic execution engine that includes electronic commerce negotiation protocols for trading stocks. MAS framework extension is also possible by adding agents that offer society wide services and providing application programmers with interface protocols that enable interaction.

There are many benefits to decomposing a MAS framework into the core capabilities that every agent must posses and those application specific features that are dictated by the problem domain or application design decisions. First, the agent research community could avoid a tremendous amount of duplicated effort if MAS frameworks were designed to interoperate with a pool of reusable components sharing a common interface. Widely useable components would encourage community wide reuse and hopefully lead to standard agent libraries whose need has been expressed but has yet to materialize. Second, there are many benefits to trimming a MAS framework so that it includes only core agent code. Application developers don't have to code around unused or unwanted code. Unused code might incur performance penalties, such as execution speed or footprint size, that could be avoided. Intuitively, it is also likely that a smaller framework (less lines of code) would be easier to learn. Third, building an agent from pluggable subsystems allows application developers to choose the best components from industry. They are no longer locked into solutions provided by MAS framework developers and are thus given a greater degree of control over customization. Finally, because this customization is at a higher level of abstraction,

application developers can construct agents from larger components thereby increasing the potential for reuse.

2.3 Agent Roles

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 Agent Work Unit layer of abstraction and can also encapsulate other lower level agent roles. Though in many MAS frameworks an agent role is a conceptual entity, it is proposed that an agent role is the smallest concrete unit of reusable agent functionality. Roles may be composed of smaller work units, representative examples of such primitives are found in Figure 1 and include tasks, graphs, nodes, and behaviors. While these components are always primitive work units, they may not be roles in and of themselves; this is subject to the application programming style and higher-level encapsulation mechanisms offered by the framework.

A protocol defines the agent roles involved in a conversation and the legal messages that can be exchanged between them. It also serves as an agent-oriented unit of encapsulation that embodies the rules of a dialogue. Because these rules "crosscut" agent roles, there is a relationship between protocols and aspects. This has implications on the decomposition of an agent. Aspect-oriented programming may help connect communication and behavior subsystems in a more maintainable and reusable manner. Further research in this area is needed but one could imagine future MAS platforms utilizing aspect technology.

The methods used to program agent behavior vary across MAS frameworks; agent roles are often not explicitly programmed and may remain a conceptual abstraction. Frameworks do provide composite components that approximate agent roles. The infrastructure provided by the framework and internals provided by these composites can range from unstructured to highly structured. Unstructured agent roles afford the programmer a high degree of flexibility but don't take advantage of factoring out reusable code across agent roles. For example, most agent roles could make use of general-purpose conversation management utilities. Highly structured agent roles are more constraining but can take advantage of built-in 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 behavior programming model. Second, there exists a tension between flexibility and reusability that must be tuned through experience.

2.4 Agent Shell

MAS frameworks typically provide agent shells that can be customized through inheritance and whose behavior can also be extended by adding agent roles. An agent composed of executing agent roles provides another unit of encapsulation by hiding their interfaces from other agents in the society. The agent layer of abstraction, where true agent-oriented programming occurs, allows agents to interact at the knowledge level using a standardized agent communication language [7]. Only within the agent are agent role methods invoked.

While MAS framework designers purport offering API's for agent-oriented programming, the current generation of MAS frameworks offer much more support for programming at the agent role level. This is evidenced by the lack of prewritten agents and agent roles. One could argue with some success that agents and agent roles are domain dependent and therefore should not be included with a general-purpose MAS. Intuition and experience suggests, however, that there must also be a collection of agent behavior patterns that solve the common tasks that agents regularly perform. Implementations of these generic patterns would accompany a MAS framework while supplemental packages extending the framework would provide domain-specific agents and agent roles. Next generation MAS frameworks will include a much richer collection of agents and agent roles. The birth and evolution of agent programming communities will give rise to an increasingly comprehensive pool of reusable agent-oriented code offered at multiple levels of abstraction including agent subsystems, agent roles, agents, and even agent societies.

3 Internal Agent Architecture Critique

In developing applications with multiple MAS frameworks, a number of architectural problems were identified. While none of the frameworks suffered from all of these problems, the extensibility and reusability of all these systems was significantly affected. Most current MAS frameworks suffer from the same root cause of these problems: the internals of an agent were not intended to interoperate with those provided by other agent frameworks or component developers. This has created groups of agent researchers and developers clustered by their preferred MAS framework that are unable to share, with a few exceptions, component libraries. Some of the characteristics that have prohibited the development and use of common component libraries are discussed.

3.1 Ad Hoc Architecture

While an agent and MAS framework can be decomposed into distinct subsystems encapsulating logically related functionality, ad hoc architectures do not adequately minimize component coupling. Components not designed for adaptation often have unnecessary dependencies, one of which is sufficient to preclude reuse. If the func-

tionality of a subsystem is accessed via many public methods that are not part of an established interface, it is very difficult to replace it with an equivalent. If a subsystem relies on MAS framework code that is not part of an established interface, then it cannot be a stand-alone reusable component. Some MAS framework designs are merely a collection of classes implementing functionality that could be organized into cohesive subsystems; classes haphazardly share many references to one another. A shallow Java package structure can indicate lack of modularization.

3.2 High Level Extensibility Mechanisms Unavailable

Most MAS frameworks provide an agent behavior extension mechanism, usually in the form of an "engine." Whether rule-based or object-based, the function of a behavior engine is to provide built-in support for programming agent behavior. Engines also typically provide a pseudo-concurrent mechanism enabling multi-tasked agent behavior execution allowing behavior primitives to be scheduled and executed without using true threads of execution.

Behavior engines make it possible to extend agent behavior but the programming granularity is too low level to be the only means used to customize agents. Often times, the behavior primitives do not encapsulate enough functionality to offer significant reuse. Building from these code fragments is difficult because the number of components used makes interface incompatibilities likely. Further, the main body of code contained in a behavior primitive is usually reachable by a narrowly defined interface of typically one or two methods whose generic signatures do not reflect the processing that is performed. This programming infrastructure encourages code to be scattered into the same method names of many classes. This effectively reduces objects to methods and disables object-oriented programming language features including specialization unless proactive measures are taken to avoid such consequences. Composite classes are often constructed from behavior primitives but unless a developer parameterizes them, the additional layer of abstraction gained is marginally useful because component functionality remains as inflexible as its internals.

As was stated, behavior engines provide some support for composite behaviors and this does help raise the programming abstraction level. While it is possible to implement a subsystem that is encapsulated within a composite or large primitive behavior, this is not a desirable solution because: 1) the functionality provided by a subsystem is more difficult to access. 2) The scheduling and process services provided by an operating system are not well utilized. 3) It is more likely that behavior execution time will not be equitably distributed.

The current generation of behavior engines does not support preemptive scheduling. They are implemented with the assumption that executing behaviors have partitioned functionality into small units of work. Behaviors are expected to act like good citizens and explicitly and expediently release the thread of control to allow other behaviors to execute. It is difficult for a subsystem to conform to such requirements without managing a pool of its own threads. Such a subsystem defeats the original purpose of limiting the number of threads an agent uses by replacing them with behaviors to avoid costly system resource consumption.

While instantiating a thread for each behavior is inefficient, it is also not a good idea to completely circumvent the scheduling capabilities that operating systems provide. The extended parts of agents constructed from current MAS frameworks often live within a single thread. MAS frameworks should support an agent architecture that balances the need to limit the number of threads with the desire to efficiently utilize operating systems services and language features. The architecture proposed in a paper to follow provides an infrastructure for composing an agent from subsystems. Subsystems have their own thread of control and manage the execution of behavior components within a single process.

Another problem with implementing a subsystem as a composite or one large primitive behavior is that its functionality is difficult to access. The behavior programming model is not geared toward allowing the other components of an agent to access active behaviors through an API. Behaviors are often short-lived objects whose functionality is accessed in response to state changes of various types, typically the reception of an incoming ACL message. Functionality, as has already been stated, is contained in nondescript methods whose signatures are dictated by the execution engine. The components of an agent have no guarantee that a behavior is available and agent architectures offer limited infrastructure to manage their availability. Some MAS frameworks give agents the ability to obtain a list of executing behaviors, other frameworks require behaviors to be in a pre-defined state. But these facilities are far from providing an easy way to coordinate behavior or gain access to the functionality they provide.

3.3 Uncoordinated or Unavailable Agent-Wide Resources

Many of the current MAS frameworks provide only the minimum amount of infrastructure required to build a multi-agent society while simultaneously providing highly specific, narrowly useful, but well-developed functionality of interest to the framework developers. Mobile agent platforms like Aglets are an illustrative example of how one capability is emphasized (mobility) while providing minimal support for the other functionality (conversation management) needed by any agent actively involved in a society. This places the burden of implementing rudimentary agent abilities on the application developer and leads to redundant code across developers and potentially across the components of an agent. For example, while some message routing code is likely required for behavior components, an agent-wide message routing service would reduce this overhead. If designed correctly, a MAS framework should provide a core set of general-purpose agent-wide services that can be utilized by framework extension subsystems or application-specific components. The average MAS framework only minimally provides such an infrastructure and these services are often not well organized. It is a worthwhile endeavor to analyze the capabilities of the large pool of available agent frameworks to determine the subset of core capabilities that a general-purpose framework should provide. Such a study would also reveal the many ways that these frameworks can be tailored and help designers plan for extension and adaptation.

Zeus is a notable exception from the average MAS framework as it provides a comprehensive set of agent-wide services that can be utilized by application programmers. A central data store, clock, event subsystem, unique id generator, and message router are but a sample of the functionality it provides. Unfortunately, these services are made available through an ad-hoc architecture that is highly coupled. Functionality is not decomposed into subsystems and that makes the systematic high-level coordination of an agent's internals impossible.

3.4 No Underlying Meta-model

(Incomplete at time of submission.)

If an agent is not modularized, then metal-models describing its functionality cannot be usefully applied. This leaves the various parts of an agent isolated from each other without any ability to reason about state or capabilities. This situation becomes worse in the context of adding agent capabilities dynamically.

- Current movement toward decentralized systems, P2P as example
- Exploiting parallelism
- Zeus pseudo meta-model based on agent-wide event system
- SYSTEMS NOT DESIGNED FOR component plug-and-play
- "subsystems" lack
    - A high level description
    - Defined states representing general service attributes
    - Application defined states
    - Ability to reason

References

[1] Agent frameworks list, www.agentbuilder.com/AgentTools

[2] Mohammed Fayad, Douglas C. Schmidt, Ralph E. Johnson, Implementing Application Frameworks, Wiley, 1999

[3] Steven P. Fonseca, Martin L. Griss, Reed Letsinger, Evaluation of the ZEUS MAS Framework, Second International Workshop in Software Agents and Workflows for Systems Interoperability, July, 2001

[4] Steven P. Fonseca, Martin L. Griss, Reed Letsinger, An Agent Mediated E-Commerce Environment for the Mobile Shopper, Hewlett-Packard Laboratories, Technical Report, HPL-2001-157

[5] Steven P. Fonseca, Martin L. Griss, Reed Letsinger, Agent Behavior Architectures, A MAS Framework Comparison, Hewlett-Packard Labs, Technical Report, HPL-2001-332, (short paper form at AAMAS 2002)

[6] Foundation for Intelligent Physical Agents. FIPA Communicative Act Specification,

PC00037E, 2000, www.fipa.org

[7] Nicholas R. Jennings, On Agent-Based Software Engineering, Artificial Intelligence, March, 2000, vol. 117, no. 2, p. 277-96