An Agent Architecture Composed from Reusable Subsystems: Decomposition Framework and High-Level Description

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 in the context of proposing a more flexible and extensible internal agent architecture than current MAS frameworks provide. This architecture provides 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. Object-oriented software development has highlighted the importance and success of building from collections of compatible components. 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 [4]. 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 architectures for connecting components together [3, 4]. 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) How can existing MAS frameworks be evolved to make agent construction easier? 2) Can an internal agent architecture be developed that facilitates better component reuse? 3) 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 [19], the framework components they are composed from are not, in general, designed for use with other MAS frameworks. Further, MAS framework and agent 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

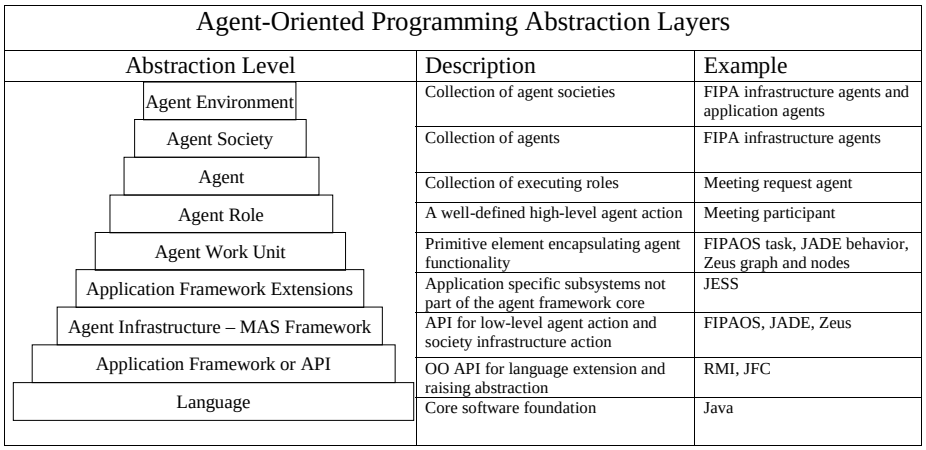


Figure 1: An agent-oriented decomposition framework.

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.

2 Agent Abstractions and Decomposition

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 as an abstraction pyramid. Higher levels of the pyramid denote a higher component abstraction level. Higher levels are shown with smaller rectangles to denote the decreasing ratio of the service interface to encapsulated functionality. The abstraction pyramid was derived after developing e-commerce and meeting scheduling agent societies [5, 6, 7, 8], synthesizing conceptual and experience-based literature [9, 10, 15, 16, 17, 21] on agent architectures, and analyzing several [2, 18, 20] MAS frameworks.

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

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 [19] and JAS [11] specifications that are on the path to extracting the essence of multi-agent system software for a partial answer.

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 exe-

cution unit. This is analogous to the JRS [14] (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 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.

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 this 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. As a side note, 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 code 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.

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 component templates for programmers to customize that vary in level of structure. The success of using these components depends on how easily agent behavior can be mapped to the internal constructs. The solution model must align with the agent behavior-programming model. Further, 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 [13] using a standardized agent communication language. 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.

3 Internal Agent Architecture Critique

In developing applications with multiple MAS frameworks, a number of architectural problems were identified. Current MAS frameworks share the 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.

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 functionality 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 but instead, 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.

Behavior engines make it possible to extend an agent 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 this paper 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

Current MAS frameworks provide little or no infrastructure for an agent to reason, respond, or monitor the execution of its constituent parts.

In a well-designed software system, interfaces are specified where change and adaptation are expected. Interfaces partition related components together into cohesive clusters whose functionality is accessed through an explicit and well-defined standard. In essence, interfaces are a form of meta-model. They allow other parts of a system to reason about functionality without knowing specific implementation details and also allow the system to have expectations that must be satisfied. If an agent was decomposed into subsystems that were accessed via standardized interfaces, then the agent could, with limited success, reason about its abilities. This would primarily offer developers the ability to interchange pieces of code and this would be an improvement to current generation MAS frameworks that make minimal use of interfaces. But object-oriented interfaces, in specifying available methods and their parameters, are much less rich in describing functionality when compared to a service description language. Further, meta-information about the agent that is not specific to any of its constituent subsystems is not available.

Another form of meta-model is an event system. While event systems do not enable component interoperability, they do provide a notification infrastructure that allows participating components to reason and respond to state changes. A contribution of the Zeus MAS framework is to demonstrate how an agent can be built from a 57-event infrastructure facilitating communication among its components. Such an event system makes a good architectural design starting point but is not a complete solution because no facilities are provided to dynamically add events and no higher-level description mechanism is available.

4 MAS2 Architectural Overview

4.1 Primary Design Requirements

- The internal agent architecture should support the decomposition framework specified in Section 2. The architecture then provides construction guidelines that lead programmers to more extensible, reusable, and modular design.
- The core services of an agent should be abstracted from all application dependent functionality and be accessed via a well-designed and standardized API that balances feature offerings with ease of use.
- Subsystems should be described by the services they provide, the services they require, and in terms of the more general characteristics of their computation.
- The current status of a subsystem should be split into general state attributes (active, registered, etc.) and subsystem specified, possibly application dependent, state attributes (conversation initiated, executing behavior x, etc.).
- Subsystems access each other's resources via an event bus that is moderated by the central intelligence of an agent and not via method execution.
- Each subsystem operates in a unique thread that is controlled by the central intelligence of an agent.
- In addition to the shared core services, each subsystem has exclusive access to a set of additional services provided by the MAS framework.

4.2 High-Level Model

Figure 2 proposes an internal agent architecture that is the synthesis of ideas shared in the decomposition framework and observations made about the architectures of current MAS frameworks. Code from a general purpose MAS framework is split into core services, central intelligence, and services offered per subsystem. Subsystems are the highest-level extensibility point of an agent and are connected via a centralized event-based software bus. A MAS framework may provide several useful default subsystems, but they are not coupled to each other and are only loosely coupled to the central agent infrastructure through established interfaces. The central agent infrastructure gives each subsystem instances of a set of useful of services. Among these services is a pseudo-concurrent scheduling mechanism for fine-grain extension of agent behavior allowing multiple behaviors to execute within its single execution thread.

4.3 Core Agent Infrastructure

The infrastructure of an agent is split into its core services, per subsystem services, and central intelligence. The core services of an agent, also thought of as the agent kernel, are the non-extraneous set of functionality useful to all agents. While accessible by the agent shell through standard method invocation, event requests and replies are used as the interface with subsystems. A determination of the services that should be offered is underway but one can look to the Zeus MAS framework for likely candidates. Representative examples include a system clock, unique identification generator, timeout notification service, event and state logging, data translators, and central storage facility.

In addition to core services, subsystems are given their own instances of a suite of services. The objective is to factor out common subsystem infrastructure. Representative services include support for event handling and maintenance, a pseudo-concurrent agent behavior scheduler, and central storage utilities.

The central intelligence of an agent contains logic for subsystem composition and control.

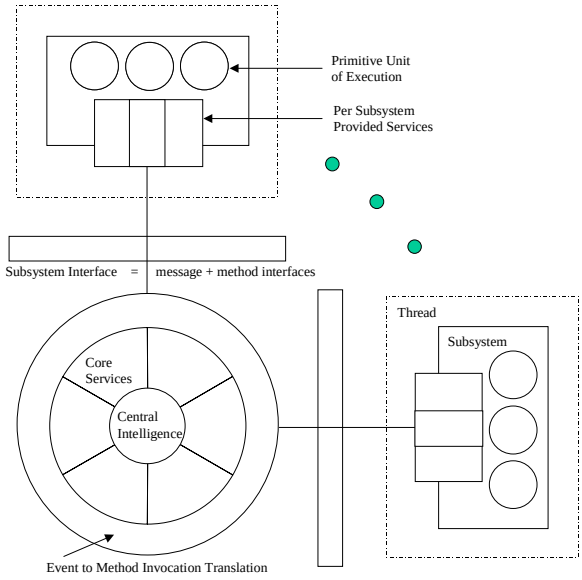


Figure 2: High-level internal agent architecture description.

4.4 Subsystems

Subsystems are added to an agent via a registration process with the agent kernel. The agent kernel determines if the subsystem is allowed to become part of the agent by ensuring the method-based subsystem interface is implemented correctly and that it

responds appropriately to system request events. After these tests, the agent kernel provides a set of descriptions for the services provided by the active subsystems. As subsystems are added, removed, and updated, the agent kernel sends corresponding notifications to all subsystems. With the registration process complete, the agent kernel starts the new subsystem thread and maintains a reference for process control and monitoring. Subsystems then maintain control of all internal execution. They have access to per-subsystem services and core services provided by the MAS framework. They are responsible for announcing internal state changes by updating a generic state description data structure. As an observer of this data structure, the agent kernel broadcasts these state changes to the entire system.

4.5 An Example Agent

A representative instantiation of the internal agent architecture of Figure 2 is shown in Figure 3. Consider the case where the behavior engine subsystem is the last of three

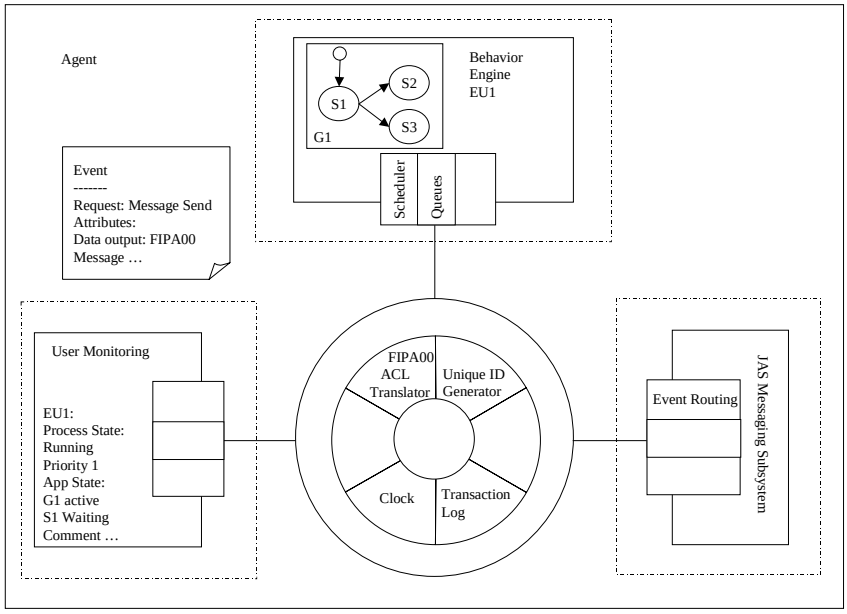


Figure 3: An instantiation of the architecture from Figure 2.

subsystems registered with the agent kernel. These subsystems provide an engine for state-based protocol execution, a JAS compliant ACL messaging service, and a GUI-based agent monitoring service for users. When the behavior engine requests to register with the agent kernel, the kernel first verifies that it is a legal subsystem. The kernel sequentially requests the behavior engine to provide its high-level service description, the attributes that characterize its application state, the events types that it will request, the event types it can service, and any purely informative events it generates. This information is centrally recorded and distributed to all active subsystems. The

subsystem then requests the per-subsystem services that it wants the agent kernel to provide. If this registration process proceeds without error, the agent kernel provides descriptions of the services provided by the messaging and monitoring subsystems.

On start-up, the behavior engine executes the first state of a predefined graph encapsulating a conversation protocol and associated strategies. In state S1, an ACL message transmittal request is sent to the agent kernel. The agent kernel transforms this request into an event. It uses the unique ID, clock (time stamping), FIPA00 message translator, and transaction log provided by the kernel to create the event that it forwards to the messaging subsystem. This event is also sent to the monitoring subsystem as when the behavior engine registered, the monitoring subsystem informed the kernel of its interest in receiving these events. In figure 3, however, the application state of the behavior engine is displayed as a result of the periodic state polling of the agent kernel.

What's not shown in the diagram is the generic internal agent architecture and subsystem ontology that defines the semantics of events and services.

5 Conclusion

Given the emergence of web services, it bears asking how this architecture supports discovery, composition, and execution. As the architecture is general-purpose and component-based, there clearly is no built-in support for utilizing web services. Subsystems are meant to reside on the same host as the core services and they are coupled by method invocation. The architecture, however, does support dynamic composition. It is therefore possible to design a subsystem that serves as an agent's interface to web services that can translate internal agent requests into external web service discovery and invocation. In fact, providing an agent with a web service manager subsystem would provide a default mechanism for handling internal agent requests that could not be handled by any other subsystem. If web services do proliferate as planned, then the functionality of a web service manager subsystem might instead be included as a core service. But web service technology must mature and truly become ubiquitous before assessing its impact on agent architectures.

Analyzing the shortcomings of current MAS frameworks demonstrates the need for developing an internal agent architecture that better supports modularity and reuse. This architecture must be founded on a decomposition framework for programming agent behavior that outlines the hierarchy of agent-oriented abstraction layers used to decompose agent functionality. Each level of abstraction should further specify the reusable components used for encapsulation and the relationships among components within and across layers. This paper proposes a decomposition framework that satisfies these requirements. After providing an analysis of the architectures provided by current MAS frameworks, a modular, event-based subsystem was described. The next step in this research is to perform a detailed design that specifies core service API's, the subsystem interface, kernel processes, event types and their descriptions, a general-purpose ontology describing the family of subsystems an agent is likely to use, and explore algorithms for intelligent subsystem composition. To validate this archi-

tecture, already available application independent subsystems will be refactored and adapted for use as subsystems.

References

1. Agent frameworks list, www.agentbuilder.com/AgentTools
2. Fabio Bellifemine et al. JADE – A FIPA-Compliant Agent Framework. Proceedings of Practical Application of Intelligent Agents and Multi-Agents. April, 1990, p. 97-108
3. Grady Booch et al., Unified Modeling Language User Guide, Addison-Wesley, 1999
4. Mohammed Fayad, Douglas C. Schmidt, Ralph E. Johnson, Implementing Application Frameworks, Wiley, 1999
5. 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
6. 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
7. 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)
8. Martin L. Griss and Reed Letsinger, Games at Work-Agent-Mediated E-Commerce Simulation, Autonomous Agents, 2000, June
9. Thomas Gschwind, Metin Ferudin, and Stefan Pleisch, ADK - Building Mobile Agents for Network and Systems Management from Reusable Components, ASA/MA. Los Alamitos, IEEE Computer Society Press, 1999. p. 13-21
10. David Harel and Michal Politi, Modeling Reactive Systems with Statecharts, McGraw Hill, 1998
11. Java Agent Services (JAS) Specification, http://jcp.org/jsr/detail/87.jsp
12. Nicholas R. Jennings An Agent-Based Approach for Building Complex Software Systems. Comm. ACM. April 2001. p. 35-41
13. Nicholas R. Jennings, On Agent-Based Software Engineering, Artificial Intelligence, March, 2000, vol. 117, no. 2, p. 277-96
14. Java Rule Engine Specification (JRS), http://jcp.org/jsr/detail/94.jsp
15. E.A. Kendall, Role Model Designs and Implementations with Aspect-oriented Programming, Proc. Of OOPSLA 99, Oct, Denver, ACM SIGPLAN, p. 353-369
16. E. A. Kendall and Krishna P.V. Murali, A Java Application Framework for Agent Based Systems, Evolve, 2000
17. R. Letsinger and E. Katz, A Flexible Architecture for Personalizable Agents, HPL Tech Report, Nov 2001. (In preparation).
18. H. Nwana, D. Nduma, L. Lee, J. Collis, ZEUS: a toolkit for building distributed multi-agent systems, in Artificial Intelligence Journal, Vol. 13, No. 1, 1999, p. 129-186.
19. P.D. O'Brien and R. Nicol, FIPA: Towards a standard for intelligent agents. BT Technical Journal, 16(3), 1998
20. Stefan Poslad et al. The FIPA-OS agent platform: Open Source for Open Standards. Nortel Networks. Manchester, UK. April, 2000.
21. Miro Samek and Paul Montgomery, State-Oriented Programming, Embedded Systems Engineering, August, 2000, p. 21-43.