A Pragmatic Agent Architecture for Layered Component Reuse Using Subsystems

Steven P. Fonseca
UC Santa Cruz, HP Labs
University of California, Santa Cruz
Santa Cruz, CA 95064, USA
1-831-234-0641
fonseca@cse.ucsc.edu

Martin L. Griss
UC Santa Cruz
University of California, Santa Cruz
Santa Cruz, CA 95064, USA
1-831-459-2502
griss@soe.ucsc.edu

ABSTRACT
Current agent toolkits make it too difficult to build large, robust agent-based applications. A key part of scaling up agent systems is the ability to quickly and reliably create individual agents, small societies of closely-related agents, and large ecosystems and clusters of these societies that share compatible properties, such as transport, security, representation, behavior models, rules and protocols. Many of these (cross-cutting) features impact the underlying agent platform, while many others have a deep impact on the structure of an individual agent. In this paper, we address a principle weakness of current multi-agent system frameworks that is the primary cause of this problem: the lack of an agent shell based on a much-more reuse oriented internal agent architecture. Such an architecture will greatly aid a developer in appropriately decomposing behavior into reusable components. This is a key to a flexible toolkit that allows the rapid construction of large numbers of compatible agents. Better conformance to the FIPA abstract architecture is a step in the right direction, but this specification does not penetrate far enough into the internal structure of an agent. Achieving the correct separation of concerns is most often left to chance, even when conceptual models, like BDI [Rao95], are used. The many problems include no guidance for choosing component granularity, insufficient core agent functionality, poor separation of general-purpose agent functionality from application specific code, and a weak component communication infrastructure. The result is that agents are crafted from code that is difficult to reuse within and across organizations. In this paper, a general internal agent architecture is proposed that is more flexible and extensible than current MAS frameworks provide. This architecture includes a component decomposition framework and infrastructure that guides application developers to decompose agent behavior in reusable ways. At the highest level, agents are constructed from reusable subsystems. 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 large-scale agent composition from industry wide best known components instead of building agents from a single MAS framework repository.

Categories and Subject Descriptors
D.2.11 [**Software Engineering**]: Software Architectures – *domain-specific architectures, data abstraction, patterns*

General Terms
Design.

Keywords
Architecture, agents, event-bus, component, framework, reuse.

1. INTRODUCTION
The agent-oriented software engineering community has not yet leveraged many of the proven object-oriented techniques for component design and construction.

MAS frameworks have been developed by many largely independent organizations with minimal informal idea cross-fertilization. However, systematic reuse calls for comprehensive analysis across the individual instances in a domain. Feature oriented domain analysis [11] and similar domain engineering techniques [5] capture the commonality and variability of a product family. Knowing the commonality and variability points of a domain is necessary to develop MAS frameworks with a reusable architecture and components that are also extensible [13]. Large-scale reuse across a community, similar to an operating system and application, requires a flexible architecture supporting extension that also provides core services that can be leveraged by application programmers. Knowing what these core services are, the range and type of desirable extensions, and a suitable architecture for connecting these components together is most expediently discovered by analyzing current MAS frameworks for design and component artifacts. There has been minimal work, mostly in the form of specifications [6, 9], to develop next-generation designs and components based on previous MAS framework experience. Further, incremental MAS framework development continues within clustered and isolated programming communities where the initial component base discourages architectural evolution.

A partial explanation for lagging MAS framework component development is a community focus on supporting agent level interoperability. It is common for agents built from different MAS frameworks to interact using a compatible communication mechanism [1] but uncommon for them to be built from compatible internal components. With the continuing

stabilization of agent communication comes the need for a shifted emphasis on component-based agent-oriented software engineering.

Component-based software engineering encourages the development of interoperable components and architectures supporting their connection because it is this consolidated community effort that leads to a reusable component infrastructure. Components designed with common interfaces across a range of functionality (security, communication, etc.) facilitate compatibility and leads to a more robust component pool because only the best instances of component types survive. Further, the community as a whole directly influences iterative and incremental development and this leads to better design and component libraries. A component-based agent-oriented engineering approach to MAS framework development and MAS applications is required to deliver scalable industrial strength software that makes deploying multi-agent societies efficient and dependable. Cost and dependability have kept agent systems from widespread deployment. Improved engineering of agent systems where generators, patterns, components, and an architecture are combined into a robust toolkit has the potential to prove agents as a reusable component [8] and agents as a solution for distributed, autonomous, and intelligent systems.

2. High-Level System Description

The internals of an agent are encapsulated in components of varying granularity to adequately balance the need for significant design and code reuse with a sufficient amount of extensibility and flexibility. In this description, a relative granularity scale is adopted; components of high granularity are small in code size and functionality, low granularity components are largest in code size and offer greater functionality, medium granularity components are somewhere in between.

Figure 1 shows a high-level view of the internal agent architecture. Agents are composed of core services, central intelligence (medium granularity) including application and infrastructure reasoning, per subsystem services, subsystems, reusable role components, and primitive work units. Core services components (high granularity) provide agent level general purpose services for subsystems and central intelligence. Similarly, subsystem service components (high granularity) provide a set of utilities for each instance of a subsystem. The central intelligence components (medium granularity) are responsible for managing subsystems, moderating the event bus that connects subsystems together, maintaining an automated state sharing process, and contain top-level application code. Subsystems (low granularity) are added to extend agent functionality and are decomposed into two types of components, reusable roles (medium granularity), and primitive work units (high granularity). The internal structure of a subsystem, however, is hidden from the other components of an agent and the implementation details are therefore left to developers.When an agent is instantiated, any required subsystems are registered. This composition decision process is included in the application intelligence of an agent. Subsystem addition and removal occurs throughout the lifetime of an agent.

An event bus is primarily used for subsystems to communicate but central intelligence also uses it on a limited basis. The events published over the bus are described by an ontology that makes it possible for infrastructure intelligence to route events according to subsystem type. This eliminates event subscription at the subsystem level. Event handlers allow event subscription and route the events received by a subsystem to waiting roles.

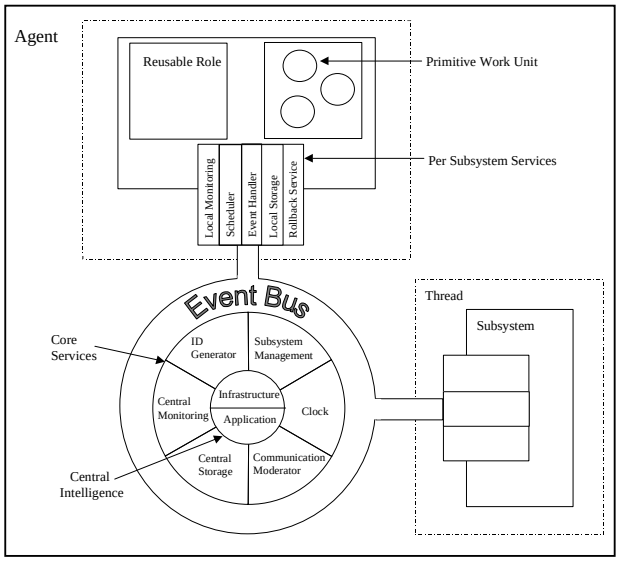

Figure 1 – High-level view of architecture.

Notice that each subsystem is contained within a single thread. This architecture balances the need to limit the number of threads used with the desire to efficiently utilize operating system services and language features. Because subsystems encapsulate a large amount of functionality, it is expected that an agent will be composed of relatively few, somewhere between 1 and 10. Following this guideline ensures that the host platform is not overwhelmed by task swapping and process management inefficiencies.

3. The Internal Agent Event System

An event-based communication bus is well suited to handle the asynchronous and dynamic environment that an agent must respond. Further, concurrent processing within an agent and the desire for loosely coupled communication between subsystems also makes a strong argument for using a software bus to connect the internals of an agent. The software bus for an agent need not be general purpose; it is more appropriate to use an automated subscription mechanism based on a general ontology describing the type of subsystems an agent is composed from and their associated capabilities. This effectively eliminates the dynamic subscriptions found in typical event bus systems by replacing them with a once per subsystem registration process. Once registered, infrastructure intelligence routes events based on the active subsystem capabilities and can further choose how an event is processed by considering any higher level goals.

Event Types. The internal agent event system is primarily used for subsystem communication. As is described in the Section 3.3, an underlying MAS framework ontology describes the type of subsystems that can be used to compose an agent. Included in this description are the events a subsystem can process, events

communicating the results of processing, and purely informative events.

3.1 Event Information Storage

While in traditional systems event information is passed to registered observers using a suite of special purpose event objects that provide methods to access event information, this architecture uses a single type of event object where information is stored using a general-purpose concept tree. The key detail in ConceptTree and EventContent classes is that hierarchically related concepts arre represented with attribute-value pairs. This structure, like XML, is suitable to express a wide variety of information and has been successfully used by a multi-agent system framework [14] to store ontologies. It is preferred for an internal agent event system because subsystems are not required to know a multiple class API to retrieve event information and eliminating this method invocation decreases system coupling. Ontology Support

Agent systems rely heavily on ontologies for agent interoperability (communication), are required for programming intelligent agent behavior, and are therefore a core agent feature supported by MAS frameworks. The average MAS framework provides built-in support for ontology description, storage and retrieval. Additional services are provided in more sophisticated framework implementations including utilities for testing concepts or their relationships. Because many components of an agent must interface with the ontology mechanism, improper (or at least, less flexible) implementations are responsible for introducing a large amount of coupling in current frameworks. The result is that agent programmers using frameworks with a different ontology representation cannot reuse components from other frameworks. Compounding this problem is the immaturity of techniques for ontology representation and management. Numerous problems remain and no unified standard exists to support cross-framework compatibility. Coupling and nonstandard ontology management explains the design decision to encapsulate this functionality within a subsystem and not as an agent core service as is equivalently done in common practice by current MAS framework designers.

While using ontologies to describe an application domain is common practice, this architecture also uses ontologies to describe the internals of an agent including how it is composed, a description of its components at multiple levels of abstraction, and dynamically updated state descriptions of internal components. This gives an agent the ability to reason about and coordinate its internal processing with much greater control than is possible with current MAS frameworks.

3.2 Internal Agent Component Description

The internal agent component ontology is composed of an event ontology, high-level subsystem ontology, and additionally describes the composition of an agent and the dynamic state of its components. State description occurs at multiple levels of abstraction including agent, subsystem, and role components. The goal of this ontology is to allow agents to reason about their internal composition and make decisions in accord with an understanding of self.

3.3 Event Description

A primary design assumption is that the core components of an agent can be organized into general subsystem types that are described by a standardized event ontology. Analyzing current MAS frameworks indicates that most agents are composed of components that can be encapsulated into distinct subsystems and it is also clear that a core set of these subsystems are virtually always present within an agent. The more difficult assumption that this architecture tests is the ability to describe subsystem functionality based on general request, response, and inform events. The design of this architecture introduces the idea of describing the internals of an agent using an event ontology, provides an example to illustrate its feasibility and potential, but must be implemented and deployed across many application domains to proof its usefulness. It is intuitive to believe that the difficulty of describing some subsystems will vary and some functionality may not be easy to generalize. A pictorial representation of an event ontology shown in Figure 2 is modeled after the Zeus event system [3]. It is not possible to show an actual ontology as that would require analyzing all of the subsystem types, extracting their features, and translating them into inform, request, and response events. Standardized API's are not even available for these subsystems, though preliminary specifications are available in some cases including coverage for the agent lifecycle [6], agent communication [9], and rule engines [10]. Instead, Figure 2 provides a comprehensive identification of subsystems and the structure used to capture their functionality. As with other event-based systems [3], events are clustered into high level categories. Representative events contained within these clusters are shown individually and could be further organized by type (request, response, or inform). A slice of the ontology is circled to exemplify how the functionality of a

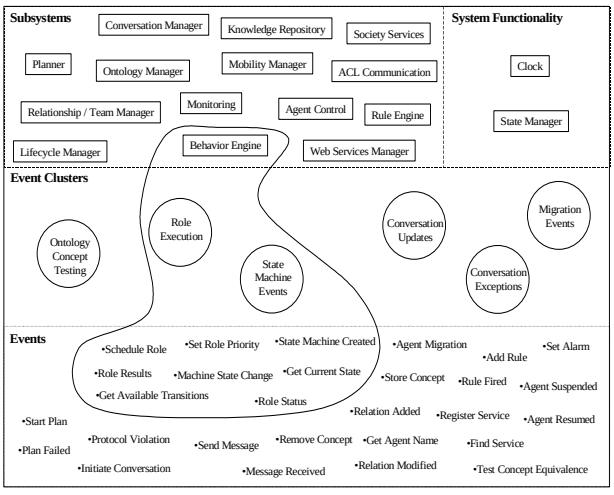

Figure 2 – The event ontology and subsystem types.

subsystem can be expressed, although more events and event clusters are needed to comprehensively specify a behavior engine.

3.4 High Level Subsystem Description

To complement the event-centric description of subsystems, a high-level subsystem ontology is used to capture non-functional

attributes. The ontology consists of attributes applicable to all subsystems and additionally defines attributes that are specific to subsystem types. Attributes used to describe a specific type of subsystem allows comparison and this gives an agent the ability to better tailor component composition to the application requirements when multiple subsystems of the same type are available at runtime. This information can also help application programmers select subsystems at design-time.

Application Domain Description. As previously discussed, ontologies for the application domain are a required agent element and the ontology management needed for their support is provided by a subsystem that is not a core service.

Subsystems. Subsystems are added to an agent via dynamic registration. Central intelligence determines if the subsystem is allowed to become part of the agent and may ensure subsystem compatibility by sending test event requests as described in Section 4.2.1 Once registered, central intelligence starts the subsystem and maintains a reference for process control and management. The subsystem shell includes the components and infrastructure shown in Figure 3. This top-level view illustrates that subsystems are composed of a high level description, events, services, state, application functionality, and have access to agent-wide services through a proxy object. Further description of these components is offered in this section.

3.5 Description

High Level Description: The high level description of a subsystem is specified by the high level subsystem ontology

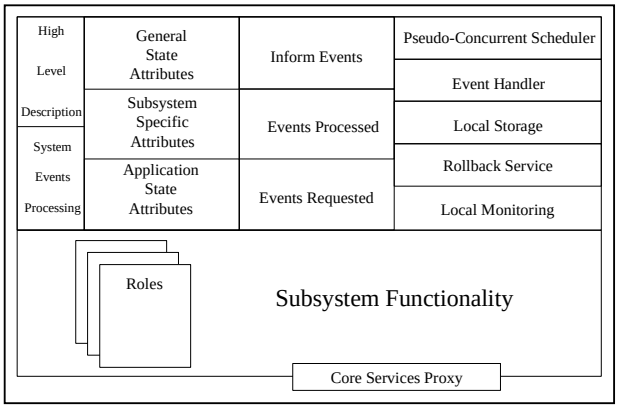

Figure 3 – Subsystem shell.

discussed in Section 3.2.

General State Attributes: General state attributes are defined in the internal agent component ontology and are applicable across subsystems of every type. For this design iteration, only the status attribute is defined but multiple attribute definitions are expected in the future. The enumeration of possible values for subsystem status includes waiting, executing, and suspended.

Subsystem Attributes: Subsystem attributes are also defined in the internal agent component ontology. These attributes are used to describe a specific type of subsystem.

Functionality Attributes: Functionality attributes are used to express application state that should be readable by the components of the agent through an event interface.

3.6 Event Set

The inform, processed, and requested event boxes shown in Figure 3 do not directly correspond to components within the subsystem object. Instead, they serve as descriptions that subsystem or agent designers can use to elicit interface requirements. These events are effectively the API of the subsystem and are fully specified by the event ontology.

System Events Processing. Architecturally compliant subsystems are required to handle all of the system events generated by infrastructure intelligence. The subsystem shell provides a component for several system event requests that automate attribute sharing. But roles responsible for processing the remaining system events must be registered with the event handler. System events are described further in Section 3.9

3.7 Per Subsystem Services

Each subsystem is given its own instances of a suite of services to factor out common subsystem infrastructure that does not belong with the core services of an agent. Because events are routed through the event handler, subsystems must instantiate and schedule at least one role to receive events from other subsystems. Other than that, subsystem developers are free to use subsystem services to the degree that is suitable for supporting their functionality. The services were designed, however, to encourage the partitioning of agent behavior into executable roles. Roles maintain references to all service objects and this facilitates role-role data sharing, scheduling roles, subsystem state saving, and event subscription.

Event Handler: The event handler routes incoming events to appropriate roles for processing. Event subscription pairs an event description with a role and the corresponding method to execute when a triggering event is received. The event description is a logical combination of descriptions that use both regular expression and event type matching. When an event is matched with a description, the event handler places the event in the role's event queue and then notifies the scheduler as shown in Figure 4. Subscriptions can be automatically removed after an event triggers or can be continuously active until an explicit unsubscribe occurs.

Pseudo-Concurrent Scheduler: The scheduler is responsible for managing role execution. Active roles are stored in two queues (Figure 4). The waiting queue contains roles requiring event reception before further processing is performed. The scheduler continuously monitors the event queues for all roles. When it pairs an event with a waiting role, the role is moved to the execution queue. Using the scheduling policy, the scheduler selects roles to execute. On selection, the triggering event is copied to the role and the method specified in the subscription is executed. The roleDone() function is used to inform the scheduler that the role has completed all processing. This function changes the role status and calls finish(). If roleDone() is not called, on completion of method execution for the triggering event, the role is returned to the waiting queue.

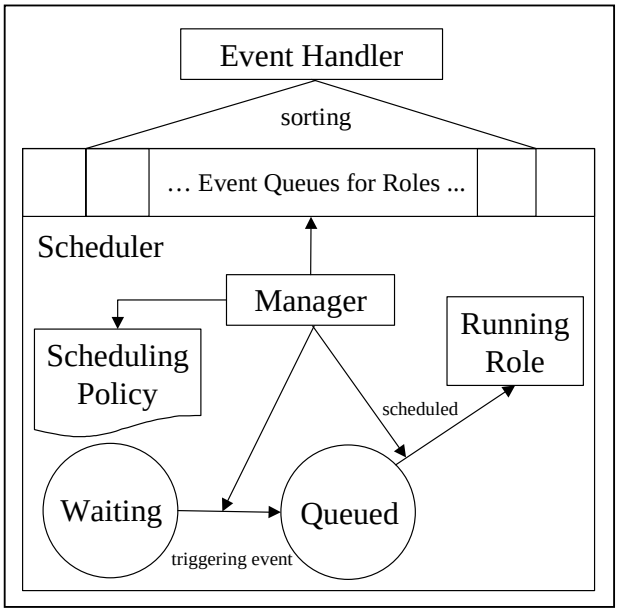

Figure 4 – Event handling and role scheduling.

Local Storage: Local storage is used to provide a blackboard data repository that facilitates role to role data exchange. Further, rollback and local monitoring use local storage to provide services including data display and snapshot reading of data for rollback.

Rollback Service: The rollback mechanism logs subsystem information at role-specified intervals allowing subsystems to rollback state for a discrete set of time instances. The information saved includes all data in local storage, general attributes, subsystem attributes, and application attributes. Future design iterations will explore also saving all role execution information including role and queue state.

·**Local Monitoring**: Local monitoring allows users to view subsystem processing. A comprehensive real-time description of subsystems is provided graphically, textually, or captured in a log file for post-execution analysis. This service works cooperatively with the other subsystem services and no programmer instruction is needed for debugging or execution display.

3.8 Core Services Proxy

The core services proxy enables subsystems to gain access to agent-wide services in a protected way that minimizes API size. Subsystem programmers use a single class to access the services that are a subset of the functionality offered to central infrastructure and application intelligence. Subsystems do not have permission to control subsystem processes, manipulate the system tick function, or configure central monitoring.

3.9 Subsystem Application Functionality

The subsystem infrastructure encourages designers to partition agent behavior into reusable roles but this is ultimately left to the subsystem developer whom can elect to use a single role as the communication medium to the agent and implement the remaining functionality in any manner.

System Processes. Subsystems are required to respond to a fixed set of system events generated by infrastructure intelligence. These events are primarily designed to facilitate the automatic sharing of real-time execution information. The request failure, subsystem addition, and system tick events are representative examples of an envisioned small set of events that a subsystem designer must provide code to handle because responses are application dependent.

Request Failure: When a subsystem broadcasts a request, infrastructure intelligence knows that a response is expected and detects when one won't be sent or a response is taking too long. The simple case is when no subsystem is available to handle the type of request broadcast. Infrastructure intelligence sends a request failure event whenever it determines a response will not be sent.

Subsystem Addition and Removal: When a subsystem is registered with the agent, infrastructure intelligence broadcasts an event describing its capabilities to all the other subsystems. For the initial design, this description is the subsystem type and its high level description. Future designs will explore subsystems encapsulating processing for a subset of the events that a given subsystem type can implement. In this case, the implemented event set must be communicated during subsystem addition. Subsystem removal events only need communicate the subsystem that is removed.

System Tick: Some agents rely on time pulses to trigger behavior in the same way as synchronous circuitry requires a clock input. When enabled, the system event tick is broadcast to all subsystems to communicate that a defined amount of time has passed.

4. Core Agent Infrastructure

The core agent infrastructure provided by a MAS framework includes components for an agent shell, core services, infrastructure intelligence, subsystem shell, and subsystem services. A placeholder class for application intelligence is also included. As subsytems and their services have been discussed, this section describes the agent shell, core services, and central intelligence. The core services of an agent are the non-extraneous set of functionality useful to all agents and their subsystems. An agent accesses these services through direct method invocation while subsystems access them through a core services proxy (proxy pattern [7]). The central intelligence of an agent is split into infrastructure and application reasoning. This reasoning is responsible for high-level agent and application management.

4.1 Core Services

An analysis of deployed agent societies, agent standards [6, 9], and several open-source MAS frameworks [2, 12, 14, 15] were used to determine the general functional commonality between agents. The result is the identification of seven core services that a general-purpose agent is likely to need.

System Clock: The system clock provides a timestamp feature and can be configured to generate internal tick events.

Timeout Service: The system clock also provides a service that generates timeout events for alarms set by a subsystem or central intelligence.

·**Unique ID Generator**: The unique identification service provides components with an agent-wide unique string generation mechanism to ensure that naming clashes do not occur between subsystems or central intelligence. Common uses for this service include generating event and conversation identifiers.

Data Translators: A data translator is an interface description for extending core agent functionality. The overhead of reading and writing event information using a general EventContent component can be minimized by providing agent-wide translation components for frequently used object representations. A subsystem can ask that event content for specific event types be automatically translated and stored in an object placed in central storage by registering for this service. In this case, infrastructure intelligence replaces the event content with a central storage key for subsequent retrieval.

Central Storage: Central storage is used to provide a blackboard data repository that facilitates subsystem to subsystem and subsystem to central intelligence data exchange.

Central Monitoring and Control: The monitoring service provides an agent-level execution view including subsystem processing summaries. Though not part of the initial API, agent control is another general service needed for design and real-time monitoring and debugging.

4.2 Central Intelligence

Infrastructure intelligence contains the logic for subsystem composition and registration, process control, event routing, and the subsystem state change notification mechanism. Application intelligence is the high level domain-dependent programming of agent behavior. It is envisioned that intelligence functionality is rule-based while subsystems are a mixture of rule and object based implementations. Subsystems and centralized components offer a two-tier separation of concerns architecture for responsibility delegation and component access control. Central intelligence components have full access to agent-wide components while subsystems control only their own local execution.

4.2.1 Infrastructure Intelligence

·Subsystem Registration, Composition, and Functionality Check: When a subsystem registers with an agent, it is agreeing to provide a set of services in an event-driven manner that is compatible with the internal architecture and corresponding component infrastructure. By design, subsystems can come from third party developers and it may not be appropriate for an agent to assume it functions correctly. In this case, infrastructure intelligence sends tester events to a subsystem to ensure that its input/output characteristics are to specification. This could be implemented as part of every subsystem registration process or infrastructure intelligence could decide when testing is required. Further, infrastructure intelligence also controls agent composition and can select the subsystems that are allowed to register; compatibility is only one factor in determining registration acceptance.

·**Process Control:** Infrastructure and application intelligence completely control all subsystem scheduling and can also remove a subsystem at their discretion. If multiple subsystems are competing for execution time, rather than allowing an operating system to determine scheduling, central intelligence can tune execution to the current operation context and agent goals. For example, processing that must be completed in real-time, perhaps resulting from a binding quality of service agreement, can be given priority over less important tasks or tasks with less demanding requirements.

Event Routing: Event routing is handled by infrastructure intelligence and is not the responsibility of a subsystem. Subsystems simply publish events to the bus and can assume it is appropriately distributed. In the case where no subsystem is available to service a request, infrastructure intelligence replies with a failure event. Infrastructure intelligence can also choose to not forward events to a subsystem if this is in the best interests of the agent. If multiple subsystems can service a request, infrastructure intelligence selects the appropriate subsystem, possibly basing its decision on their high level description or the current values of their attributes.

Subsystem State Management: Infrastructure intelligence provides an automated subsystem state sharing mechanism that keeps all agent components informed of state changes. The mechanism can receive attribute update events sent proactively by a subsystem and also initiates attribute request cycles to maintain a current view of subsystem processing. Central intelligence configures this event-sharing behavior where event policy decisions made by application intelligence override that of infrastructure intelligence.

4.2.2 Application Intelligence

The internal agent architecture includes a placeholder for centralized application code. This code has full access to all core services and the agent component. Application intelligence encapsulates higher-level behavior, conceptually it manages a collection of roles for a particular application domain. Drawing an analogy from computer hardware design, application intelligence is the CPU, subsystems are PCI cards, subsystem functionality represent on card chips, subsystem services are card controllers, infrastructure intelligence are the motherboard and bus controllers, and core services are CPU processing abilities.

5. Example Transaction

A representative instantiation of the internal agent architecture is shown in Figure 5. Consider the case where an agent is composed of a behavior engine subsystem and a JAS compliant ACL messaging subsystem. Subsystems were added by application intelligence just after the agent was instantiated and central monitoring was also set to text-based output at this time. Application intelligence broadcasts a schedule event role that is routed by infrastructure intelligence to the behavior engine subsystem. The behavior subsystem schedules the role. A state machine representing an auction conversation protocol and associated processing executes state S1. This state requires the transmission an ACL message. The role broadcasts an ACL message send event with the message in the event content and waits for a response. Previously, the JAS subsystem registered to have the content of message send event types translated into a performative object. Once central intelligence determines the receiving subsystem, the event content is replaced with a central storage key to retrieve the performative stored by a FIPA00 language translator (not shown in the core service ring in Figure 5). The JAS subsystem event handler receives the send message

event and routes this to the messageSend role. This role subsequently retrieves the performative from storage and processes the request. Throughout the course of this interaction, central monitoring provides textual reports of agent activity. Figure 5 provides a representative view at the time when the send message event is processed.

6. Concluding Remarks

Analyzing the shortcomings of current MAS frameworks demonstrates the need for developing an internal agent

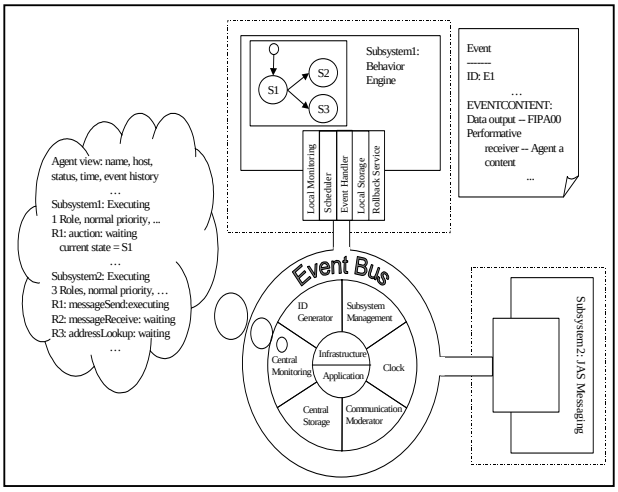

Figure 5 – Example: an instantiated architecture.

architecture that better supports component-based software engineering of agent systems where modularity and reuse are key design requirements. This is an essential to provie platforms and agent construction kits that support consistent, large-scale agent societies. The first wave of MAS framework development has completed and while some researchers argue that this topic is passé, the fact remains that an industrial strength general-purpose MAS framework has still not emerged. Further, the multi-agent system application development community does not have a reusable component base. As electronic services technology matures, the distributed systems community is likely to shift effort to developing service clients. Agents are a well suited paradigm for this problem space but without robust platforms this technology base is in jeopardy of being replaced by new solutions from the distributed solutions community – a community that is more grounded in developing large-scale industrial strength software.

References

[1] Agentcities, www.agentcities.org

[2] Bellifemine, Fabio et al. JADE – A FIPA-Compliant Agent Framework. Proceedings of Practical Application of Intelligent Agents and Multi-Agents. April 1999. p. 97-108

[3] Jaron Collis. The Zeus Technical Manual. British Telecom, BT Labs, 1999

[4] RS. Cost, T. Finin, Y. Labrou, X. Luan, Y. Peng, I. Soboroff, J. Mayfield & A. Boughannam, "Jackal: A Java-Based Tool for Agent Development." Working Notes of the Workshop on Tools for Developing Agents (AAAI '98) (AAAI Technical Report).

[5] K. Czarnecki, U.W. Eisenecker, Generative Programming, Addison-Wesley, Canada, 2000

[6] Foundation for Intelligent Physical Agents, www.fipa.org

[7] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, “Design patterns,” Addison-Wesley, 1995.

[8] Martin L. Griss and Robert R. Kessler. “Achieving the Promise of Reuse with Agent Components.” First International Workshop on Software Engineering for Large-Scale Multi-Agent Systems, ICSE, May 2002, p. 1-9.

[9] Java Agent Services (JAS) Specification, http://jcp.org/jsr/detail/87.jsp

[10] Java Rule Engine Specification (JRS), http://jcp.org/jsr/detail/94.jsp

[11] Kang, K., et al. Feature-Oriented Domain Analysis (FODA) Feasibility Study (CMU/SEI-90-TR-21, ADA 235785). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1990.

[12] Danny B. Lange and Mitsuru Oshima. “Programming and Deploying Java Mobile Agents with Aglets”, Addison-Wesley, 1998

[13] Jacques Meekel, Thomas B. Horton, Robert B. France, Charlie Mellone, Sajid Davi. “From Domain Models to Architecture Frameworks.” Proceedings of the Symposium on Software Reusability, 1997, p. 75-80.

[14] 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, pp. 129-186. (See http://www.labs.bt.com/projects/agents/zeus/).

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

[16] Anand S. Rao and Michael P. Georgeff, BDI Agents: From Theory to Practice, Proceedings of the First International Conference on Multi-agent Systems (ICMAS-95), San Francisco, June 1995, p. 312-319

Columns on Last Page Should Be Made As Close As Possible to Equal Length