Development and Specification of a Reference Architecture for Agent-Based Systems

William C. Regli, *Senior Member, IEEE,* Israel Mayk, *Senior Member, IEEE,* Christopher T. Cannon, Joseph B. Kopena, Robert N. Lass, *Student Member, IEEE,* William M. Mongan, *Member, IEEE,* Duc N. Nguyen, Jeff K. Salvage, Evan A. Sultanik, *Member, IEEE,* and Kyle Usbeck

***Abstract*—The recent growth of agent-based software systems was achieved without the development of a reference architecture. From a software engineering standpoint, a reference architecture is necessary to compare, evaluate, and integrate past, current, and future agent-based software systems. The agent systems reference architecture (ASRA) advances the agent-based system development process by providing a set of key interaction patterns for functional areas that exist between the layers and protocols of agent-based systems. Furthermore, the ASRA identifies the points for interoperability between agent-based systems and increases the level of discussion when referring to agent-based systems. This paper presents methodology, grounded in software forensics, to develop the ASRA and provides an overview of the resulting architectural representation. The methodology uses an approach based on software engineering techniques adapted to study agent frameworks—the libraries and tools for building agent systems. The resulting ASRA can serve as an abstract representation of the components necessary for facilitating comparison, integration, and interoperation of software systems composed of agents.**

***Index Terms*—Agents, distributed artificial intelligence (AI), multiagent, reference model, reverse engineering, software engineering, software architecture.**

I. Introduction

AGENT-BASED approaches for constructing complex distributed systems provide advantages over traditional software and system design methods [1]–[3]. Unfortunately, the software industry has been slow in adopting agent-oriented paradigms. It is believed that this may be partially due to the lack of integration, interoperation, and general-purpose technologies [4]. Standards bodies such as the Foundation for Intelligent Physical Agents (FIPA) [5]–[7] and IEEE are leading efforts to standardize protocols and formats of an agent-based system to facilitate interoperation of agents within agent-based systems. However, a comprehensive reference architecture describing interactions among agents and the infrastructure upon which agents execute does not exist. Such a reference architecture would increase value among agent framework developers, agent system developers and architects, and further promote the use of agent-oriented methodologies.

This paper presents an agent systems reference architecture (ASRA) that describes infrastructure-level architectural software patterns between functional components of an agent-based software system. The paper is a product of a team of interdisciplinary researchers working under the aegis of the U.S. Army's Networking Integrated Process/Product Team (IPT) subteam on intelligent agents—a group that included active participation of over fifty government, academic and industrial researchers over a four year period.

The resulting ASRA is a set of architectural view documents, each representing the design considerations from the perspective of particular stakeholders, in particular: 1) agent framework designers, those developing the libraries and agent middleware; 2) agent system developers and agent system designers, those building systems based on or using agent-based software; and 3) agent standards designers [8], [9]. The architectural views of the ASRA are constructed using a serial approach for identifying the interaction patterns among several common agent system frameworks.

A unique contribution of this paper is the use of a forensically-grounded methodology based on software engineering analysis to identify a consensus reference architecture. By using software forensics across a number of widely used agent framework implementations, the resulting ASRA is representative of the current generation of agent framework implementations. This differs from (and complements) recent and related work aimed at defining and improving software engineering practice for agent-based systems [10], [11], [13], [14] in a number of ways. Most significantly, in the forensically-grounded approach, we introduce a quantitative methodology for assessing the adherence of an implementation to a prescribed architecture. Additionally, the results of our study objectively substantiate alternative approaches of a more theoretical or abstract design nature [15], [16].

Manuscript received October 25, 2011; revised November 1, 2012 and April 12, 2013; accepted April 26, 2013. Date of publication November 13, 2013; date of current version January 13, 2014. This paper was recommended by Associate Editor L. B. Sheremetov.

W. C. Regli, C. T. Cannon, J.B. Kopena, R. N. Lass, W. M. Mongan, D. N. Nguygen, J. K. Salvage, and E. A. Sultanik are with Drexel University, Philadelphia, PA 19104 USA (e-mail: regli@drexel.edu; ctc82@drexel.edu; jbk23@drexel.edu; rnl22@drexel.edu; wmm24@drexel.edu).

I. Mayk is with the U.S. Army Research, Development and Engineering Command, Communications-Electronics Research, Development and Engineering Center, Aberdeen Proving Ground, Aberdeen, MD 21001 USA (e-mail: israel.mayk@us.army.mil).

K. Usbeck is with Raytheon BBN Technologies, Cambridge, MA 02138 USA (e-mail: kusbeck@bbn.com).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TSMCC.2013.2263132

The ASRA enables comparison and analysis of software systems composed of heterogeneous agents by providing a set of interaction patterns between functional components present in agent systems. Moreover, the ASRA facilitates interoperation by highlighting system integration points. Other contributions of this paper include:

1) illustration of a serial approach for creating and documenting a reference architecture based on a domain reference model (a reference model for agent-based systems was presented previously in [17], [18]);
2) presentation of concrete architectures mapping specific components of representative agent frameworks to functional components present in agent systems;
3) identification of similar and contrasting functional component architectures among agent frameworks;
4) description of an abstract architecture for agent frameworks as a set of patterns for functional components of agent system architectures.

The primary contributions of this paper are the process for creating the ASRA and the resulting ASRA. The serial process uses software engineering analysis techniques and architecture documentation models to derive concrete architectural representations of functional concepts for agent systems. The resulting concrete architectures for each functional concept is compared to identify abstract patterns across the functional concepts. The second result of our research is the reference architecture for agent-based systems. The ASRA enables comparison and interoperation of agent-based systems through identified patterns and paradigms of functional concepts found in such systems.

The rest of this paper is organized as follows. Section II provides a survey of reference architectures and relevant systems, including our definition of a reference architecture for agent-based systems. Section III describes the serial approach for creating the ASRA with an example application of this approach. Section IV gives an overview of the ASRA and Section V provides some examples of how to make use of it in practical situations. Section VI concludes the paper.

II. Background

This section provides background on software reference architectures and their relationship to a software reference model; it also gives an overview of related reference architectures and systems.

A. What Is Reference Architecture?

A reference architecture is a set of abstract architectural elements of a family of systems or components independent of specific technologies, protocols, and products [19]. A reference architecture describes patterns of how components of a software system interact. A reference architecture drives the creation of multiple designs for a particular system.

The consensus for describing the reference architecture is through the use of standardized diagrams [e.g., UML and other architecture description languages (ADLs)], and describing the architecture through different viewpoints to cover the concerns of stakeholders in the system [20]. UML [21] diagrams for reference architectures abstract the implementation details of a system and illustrate the relationships between the components of a system [22]. The recent effort of Sturm *et al.* [13] creates a similar methodology for agent-based systems that is highly complimentary to the ARSA. Where ARSA quantitatively grounds the different levels and structures in the overall architecture for a heterogeneous agent system, their methods provide improved formality at the design and implementation level. Their paper has a case study with a fruit market scenario that is similar to the detail case studies we performed. These are described briefly in Section V.

What is a Reference Model? A reference model describes the abstract functional elements of a system. A reference model does not impose specific design decisions on a system designer. APIs, protocols, encodings, etc. are standards that can be used concurrently with a reference model. A reference model does not define an architecture; rather, a reference model can drive the implementation of multiple architectures in the same way a reference architecture could drive multiple designs, or a design could drive multiple implementations. The reference model provides a common ontology, innovative and practical system engineering techniques, and software development guidance [17], [18].

A reference model differs from a reference architecture in a way that a reference architecture further describes the relationships and patterns of the functional elements and components of a system. An example reference model is the ISO Open Standards Interconnection (OSI) reference model [23]. This model representing the seven layers of the network describes only the functional layers of the network and does not impose a specific protocol or implementation. Existing protocols such as TCP/IP, and Appletalk fit in this model or one can create their own for each layer.

B. Other Work on Reference Architecture

In the software engineering community, architecture is an abstract representation of a software system. The architecture is composed of structures and components of the system, their properties, and the relationships between them [20], [24]–[26]. A reference architecture for software systems are the variations of relationships among components based on a reference model [25]. Collins-Cope and Matthews [26] examine the relationship between the reference model and a reference architecture illustrating the relationships between layers comprising a reference model of object-oriented/component-based systems. These relationships between objects and components form the reference architecture.

IEEE 1471 [24] states that a software architecture is the definition and relationships of a software system's components, subsystems, and interfaces for a particular set of stakeholders—individuals or entities with interest to some aspect of the software system. IEEE 1471 emphasizes each stakeholder's set of concerns in the system architecture; architectural views of the system address these concerns. Therefore, the system's architecture is a set of architectural views that address the stakeholders' concerns. The IEEE 1471 standard is based on the rational/4+1 view model [8], [9] depicting a software systems architecture using multiple architectural descriptions, or views, for stakeholders. The views in the rational/4+1 view model are as follows.

1) The logical view describes the static structural layout of the software system from the perspective of a software developer.
2) The process view describes the runtime behavior of the system, including concurrency relationships and ordered tasks carried out by components of the system from the perspective of a workflow designer or manager.
3) The implementation view describes the package layout of the system from the perspective of the system architect.
4) The deployment view describes the hardware-software configurations at a platform-level as viewed by system administrators or deployment teams.
5) The scenario view is composed of narrative use cases to provide an executive level view of the architecture. This view crosscuts the other views by providing an overall picture of the system useful for all stakeholders.

Other reference architectures also adopt presented architectural descriptions as a set of view documents. The Reference Architecture Foundation for Service Oriented Architectures (RAF-SOA) [19] from the OASIS Foundation[1] defines a reference architecture as the set of concepts and relationships defined by the OASIS reference model for service-oriented architectures. The RAF-SOA provides a template solution for which one type of SOA may be constructed. The RAF-SOA provides multiple viewpoints. Each viewpoint has a set of main concepts addressed in the architecture, the stakeholders, goals for stakeholders by using the architecture, and the modeling techniques for each viewpoint in the architecture. The RAF-SOA addresses a SOA from a Service Ecosystem, a Realizing SOA, and an Owning SOA viewpoint. These high-level to low-level viewpoints cover concerns for stakeholders with various business needs.

Within the agent-based systems literature, Weyns and Holvoet's Reference Architecture for Situated Multiagent Systems [27] focuses on the agent operating in an application environment. This architecture was developed through an iterative process of analysis and validation studying different agent-based systems. In their reference architecture, the authors constructed multiple documents from different views: the module decomposition, the shared data, and the communicating processes views. The reference architecture offered in this paper is distinguishable from this prior art due to its basis on source code analysis of fielded agent software.

The need for an ARSA is also clear from recent trends in the literature. Gholami *et al.* [28] surveyed agent-based systems as middleware infrastructure to support a wide variety of applications. While this represents a more restricted view than taken in the ASRA, their findings are consistent with ours. Khalgui and Hanisch [10] examine communications protocols among agents—work that is relevant for the design and implementation of the messaging architectures present in the ASRA. Vokřínek *et al.* [11] look at the infrastructure needs for coordination and distributed planning among agents, work that is relevant the design and implementation of the functional concepts of the ASRA, including the implementation of the agents themselves as well as how agents communicate and manage conflicts. Seow *et al.* [12] offers work in a similar vein, but focused on BDI agents rather than on distributed planning.

Approaches for agent-oriented modeling, such as that described in [14], also can benefit from the ASRA. Specifically, Cao *et al.* [14] described an improved software engineering approach to design of an overall agent-based system. This is in contrast to ASRA, which focuses on the architecture for the framework and infrastructure on which these agents will be implemented.

C. Methods of Building Reference Architectures

A survey of methodologies and processes for creating reference architectures yields few processes grounded in forensic analysis of software systems. Most work describing reference architectures for various systems [26], [29] do not, however, address the process and methodology for creating the reference architecture. Research on the methodology for creating a reference architecture is progressing. Eixelsberger's case study on recovering a reference architecture from an existing system uses an architecture structure description language (ASDL) and ADL to find commonalities of a system family [30]. This case study acknowledges that a reference model drives the creation of reference architectures and compares the ADL of instantiated systems to create a reference architecture in ASDL for the purpose of creating new systems.

The product line software engineering, domain-specific software architecture (PuLSE-DSSA) [31] is a process for creating reference architectures in an iterative fashion. PuLSE-DSSA constructs a reference architecture by: 1) generating scenarios from requirements; 2) categorizing the scenarios based on variability, structure, and priority; 3) developing architectures for the structure-based scenarios; 4) ranking the architectures based on coverage of scenarios; and 5) repeat as necessary until all scenarios are covered. PuLSE-DSSA is an iterative process that yields a reference architecture from instantiated architectures.

The real-time control system is a reference architecture for hierarchical intelligent control. The RCS was a basis for a number of reference architectures. Albus [32] describes a reference model for intelligent systems, including the controllers of agents. The architecture was adapted by NASA for the construction of telerobotic controllers [33] and many other organizations.

III. Constructing the ASRA

The process for creating a reference architecture for agent system frameworks starts with reference model functional concepts and iteratively applies software analysis tools on agent framework implementations to aid in creating architectural views of the agent frameworks. This iterative process yields four views, the scenario view, the process view, the logical view, and the implementation view. These views provide an abstract representation of the canonical patterns and paradigms present in agent framework implementations and comprise the

[1]Available at http://www.oasis-open.org.

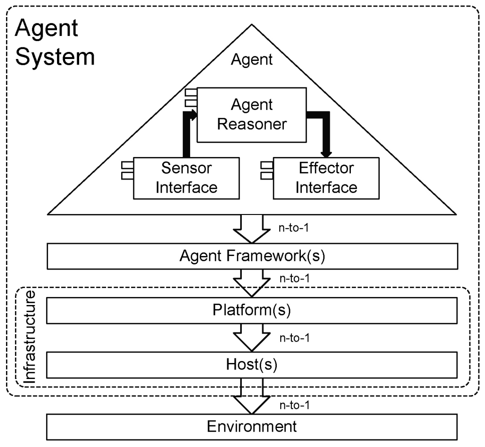


Fig. 1. Layered agent systems reference architecture model.

ASRA. The initial steps in this process rely on functional concepts provided by the agent systems reference model (ASRM) [17], [18]. Constructing the views from these functional concepts require analyzing existing agent framework implementations. The rest of this section provides a brief description of the ASRM, explaining the criteria for selecting the agent framework implementations, defining the scope of the ASRA and the process for creating the ASRA.

A. Agent Systems Reference Model

The basis for constructing the ASRA is the ASRM [17], [18], a description of software systems composed of agents. The ASRM established terms, concepts, and definitions needed for the comparison of agent systems. The ASRM also formalized concepts and layers of organization in an agent-based system. The layers, shown in Fig. 1, are: agents, frameworks, platforms, hosts, and environments. An agent-based system is the set of frameworks, the agents that execute in them, the platform (other software, OS, etc.) that supports them and the hosts (hardware) upon which they execute. The functional concepts of an agent system support overall system execution. They are defined as follows.

1) Agent administration facilitates and enables command and control of agents and allocates resources to those agents as needed.
2) Security and survivability prevents execution of undesirable actions within an agent system while allowing execution of desirable actions.
3) Mobility facilitates and enables the migration of agents among framework instances (typically, but not necessarily, on different hosts).
4) Conflict management facilitates and enables the management of interdependencies between agents activities and decisions.
5) Messaging facilitates and enables information and data transfer among agents in the system.
6) Logging facilitates and enables information about events to be recorded occurring during system execution for subsequent inspection.
7) Directory services facilitate and enables the locating and accessing of shared resources.

The functional concepts are necessary in developing the ASRA as they are the starting point for the analysis process.

B. Scope of Agent Systems Reference Architecture

The intent of the ASRA is not to define specific implementation details of an agent system, but rather to describe potential interactions between the ASRM functional concepts. The interactions between the functional concepts can be described from multiple standpoints.

1) An agent-internals architecture describes the software structure of the control-sense-effect mechanisms of an individual agent.
2) An agent framework architecture describes the software architecture and the functional components of an agent framework.
3) An agent-group architecture describes the structure and interaction of a set of agents in a group of one or more agents. The agent group architecture covers the definitions of agent team, agent organization, agent society, and agent agency.
4) An agent-systems interaction architecture describes the structure and interaction of agents with other non-agent software components within a larger system.

This paper focuses on the agent framework architecture. The internals of an agent has been studied extensively and furthermore has such wide diversity that creating a reference architecture for internals would not be useful universally. The agent-group architecture also is dependent on the domain and applications of the agent system. Agent-system interaction architectures are not directly addressed in this paper as integration and interaction of agent systems with external systems is an engineering challenge addressed by systems research and web service composition.

C. Serial Process for Architecture Discovery

The goal of the serial process is to produce overlapping series of documents and diagrams detailing many views of a system from different perspectives.

Agent frameworks analyzed: The serial process uses existing agent framework implementations to produce the architectural views of the ASRA [34]. As such, the ASRA methodology examines three open source, mature agent frameworks written in the Java language. Jade [35] aims to adhere to FIPA specifications and standards and is used in research and industrial applications. AGLOBE [36] is an agent framework that facilitates agent communications focused on experimental scenarios. It is used for a variety of applications such as airline flight planning simulations [3]. Cougaar's [2] design goal was to support tasks, workflows, expansion, and aggregation in the planning logistics domain.

During the course of this paper, the team examined over a dozen different agent frameworks, eventually settling on these three as an representative sample for detailed analysis for a number of reasons. First, these frameworks are mature and used in fielded agent-based systems with complete source code available. Source code availability facilitates better analysis and validation through source code inspection. Second, each

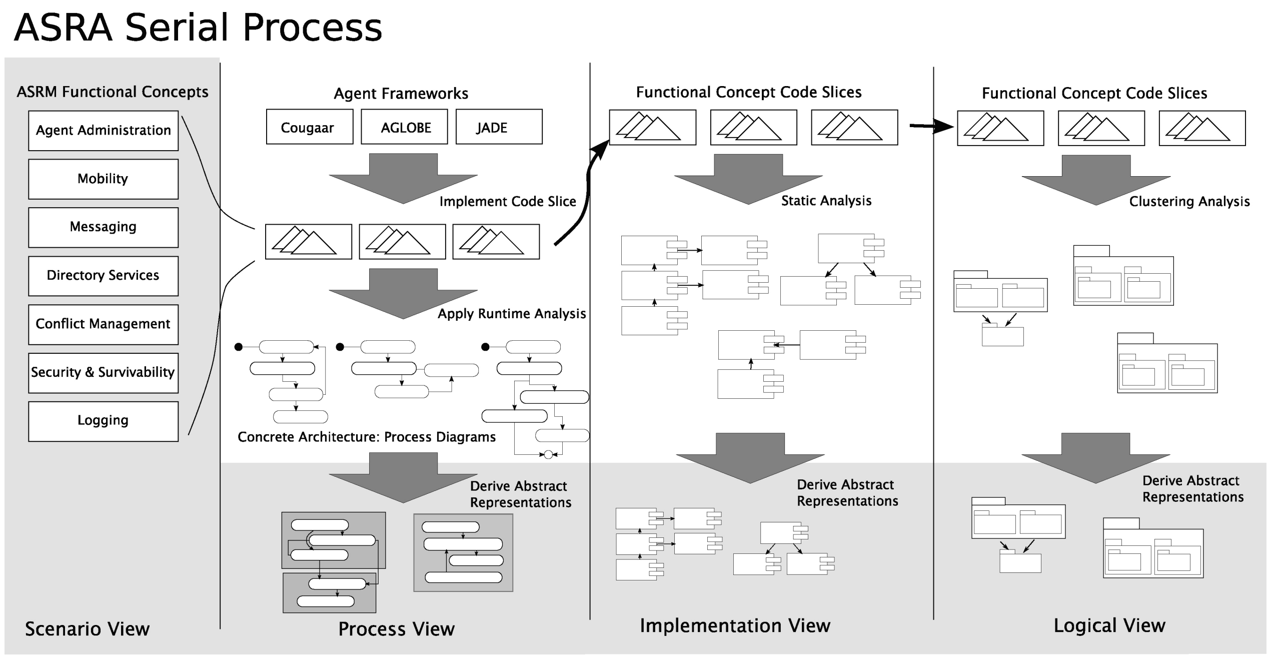


Fig. 2. ASRA documentation process for constructing a reference architecture.

agent framework embodies most of the ASRM functional concepts minimizing the number of frameworks under review. Third, the process for creating the ASRA uses tools that inspect software written in the Java programming language requiring the agent frameworks under analysis to be written in Java. Finally, each agent framework vary in design goals and implementation. Jade and AGLOBE treat agents as a computational process that maps almost directly to an executing Java thread. Cougaar's design goals lead to a different implementation with emphasis on a common blackboard for agent interactions.

For each functional concept defined in the ASRM apply the following process illustrated in Fig. 2.

1) Construct the scenario view for a functional concept. The scenario view consists of functional concept definitions from the ASRM represented as UML use-case diagrams and/or descriptions depicting the use, role, and functionality of the concept.
2) Construct the process view from the scenario view. To illustrate runtime processes, a snippet of code exercising the functional concept for each agent framework is created. Execute this snippet of code and use the dynamic runtime analysis framework, Enterprise Java Profiler (EJP),[2] to generate trace data. With this trace data, construct a UML process diagram to illustrate a concrete architecture for the functional concept for a particular agent framework. After constructing process diagrams for each agent framework, create a new process diagram from the common features across the agent framework implementations while documenting differences as points of variation. This abstract architecture for the functional concept and the points for variation comprise the process view.
3) Construct the implementation view using the static analysis tools, BAT [37] on the agent frameworks and code snippets from Step 2 to identify data flow and package/class dependencies of each functional concept. Focusing on the code snippets allows one to bypass extraneous information such as dead code and common library dependencies. From the package/class dependencies, UML component diagrams are constructed to depict concrete architectures for each agent framework functional concept implementation. Components represent the modules and packages and connectors represent interdependencies. From the concrete architectures, an abstract architectural package representation is created by identifying similar packages and modules across implementations. Different packages are documented as points of variation.
4) Construct the logical view using the Bunch clustering system [38] and the static analysis data from the previous step. The logical view consists of UML package diagrams of a functional concept. This abstract architectural representation of a functional concept is created from the concrete architectural views of the agent frameworks. The clustered data, represented as a graph, illustrates interdependencies between components (edges), and modules (nodes) within the agent framework implementation. Highly-connected modules indicate components and subsystems within an agent framework implementation. UML package diagrams depict the the concrete logical architecture of each agent framework implementation where packages are the modules and the connectors are interdependencies. Packages within other packages represent interdependencies that do not travel

[2]Available at http://ejp.sourceforge.net.

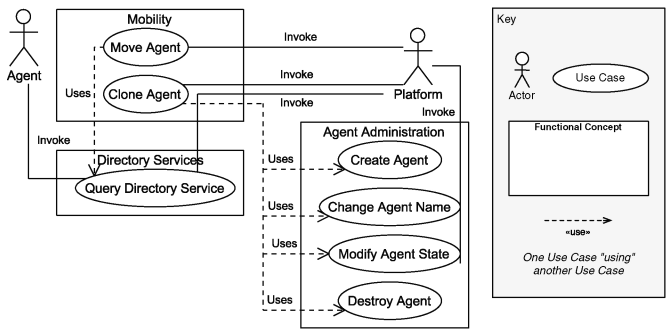


Fig. 3. Mobility functional concept use case, showing interactions with administration and directory services.

outside the enclosing package. From the concrete logical architectures of the agent frameworks, an abstraction of the logical package diagrams are created noting similarities and differences where differences are documented.

The four steps are repeatedly applied for the seven ASRM functional concepts. The resulting architectural views for the functional concepts comprise the ASRA.

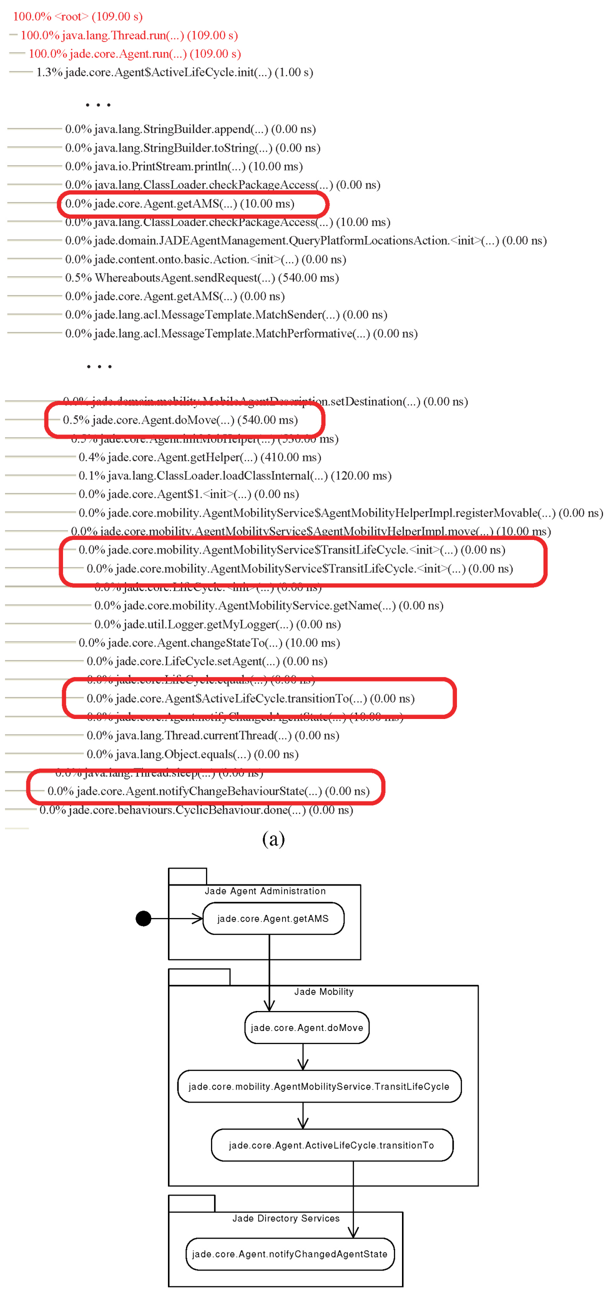
D. Demonstration of the Serial Process

To illustrate this mechanism for developing the reference architecture, we provide a detailed example of how to apply this serial process to understand how different frameworks have implemented the Mobility functional concept. This process is then repeated for each of the other functional concepts, the complete details of which are available in [39].

1) *Construct the Scenario View for Mobility:* The scenario view for each functional concept consists of UML use-case diagrams and/or descriptions based on the ASRM definitions. Mobility is the process by which the serialized, nonexecuting agent leaves the source framework instance and arrives at a destination framework instance. The mobility functional concept is described by the following processes: decision procedure, either active or passive; deregistration, halting, serialization of the agent; migration; and deserialize, reregister, and resume.

The UML use case diagram in Fig. 3 depicts the two basic uses cases (ellipses) for the mobility functional concept: moving an agent and cloning an agent. The move agent use case entails moving an agent from one container to another, while the clone agent makes a copy of an agent in another container. The platform actor invokes these use cases. Note, this figure also illustrates interactions between the Agent Administration and Directory Services functional concepts. For example, the clone agent use case utilizes the create agent and modify agent state use case.

2) *Construct the Process View for Mobility:* The process view documents the runtime behavior of a functional concept based on code snippets for each agent framework (Fig. 4). Executing EJP on code snippets yield runtime traces (Fig. 4) showing a temporal view of the mobility functional concept, the invocation points of the mobility functional concept, and the percentage of time spent in the methods during execution. From the runtime trace, a direct mapping to a UML activity


```
100.0% <root> (109.00 s)
 — 100.0% java.lang.Thread.run(...) (109.00 s)
 —— 100.0% jade.core.Agent.run(...) (109.00 s)
 ——— 1.3% jade.core.Agent$ActiveLifeCycle.init(...) (1.00 s)

          . . .

 ———————— 0.0% java.lang.StringBuilder.append(...) (0.00 ns)
 ———————— 0.0% java.lang.StringBuilder.toString(...) (0.00 ns)
 ———————— 0.0% java.io.PrintStream.println(...) (10.00 ms)
 ———————— 0.0% java.lang.ClassLoader.checkPackageAccess(...) (0.00 ns)
 ———————— 0.0% jade.core.Agent.getAMS(...) (10.00 ms)
 ———————— 0.0% java.lang.ClassLoader.checkPackageAccess(...) (10.00 ms)
 ———————— 0.0% jade.domain.JADEAgentManagement.QueryPlatformLocationsAction.<init>(...) (0.00 ns)
 ———————— 0.0% jade.content.onto.basic.Action.<init>(...) (0.00 ns)
 ———————— 0.5% WhereaboutsAgent.sendRequest(...) (540.00 ms)
 ———————— 0.0% jade.core.Agent.getAMS(...) (0.00 ns)
 ———————— 0.0% jade.lang.acl.MessageTemplate.MatchSender(...) (0.00 ns)
 ———————— 0.0% jade.lang.acl.MessageTemplate.MatchPerformative(...) (0.00 ns)

          . . .

 ———————— 0.0% jade.domain.mobility.MobileAgentDescription.setDestination(...) (0.00 ns)
 ———————— 0.5% jade.core.Agent.doMove(...) (540.00 ms)
 ————————— 0.5% jade.core.Agent.initMobHelper(...) (530.00 ms)
 —————————— 0.4% jade.core.Agent.getHelper(...) (410.00 ms)
 —————————— 0.1% java.lang.ClassLoader.loadClassInternal(...) (120.00 ms)
 —————————— 0.0% jade.core.Agent$1.<init>(...) (0.00 ns)
 —————————— 0.0% jade.core.mobility.AgentMobilityService$AgentMobilityHelperImpl.registerMovable(...) (0.00 ns)
 ————————— 0.0% jade.core.mobility.AgentMobilityService$AgentMobilityHelperImpl.move(...) (10.00 ms)
 —————————— 0.0% jade.core.mobility.AgentMobilityService$TransitLifeCycle.<init>(...) (0.00 ns)
 ——————————— 0.0% jade.core.mobility.AgentMobilityService$TransitLifeCycle.<init>(...) (0.00 ns)
 ———————————— 0.0% jade.core.LifeCycle.<init>(...) (0.00 ns)
 ———————————— 0.0% jade.core.mobility.AgentMobilityService.getName(...) (0.00 ns)
 ———————————— 0.0% jade.util.Logger.getMyLogger(...) (0.00 ns)
 —————————— 0.0% jade.core.Agent.changeStateTo(...) (10.00 ms)
 ——————————— 0.0% jade.core.LifeCycle.setAgent(...) (0.00 ns)
 ——————————— 0.0% jade.core.LifeCycle.equals(...) (0.00 ns)
 ——————————— 0.0% jade.core.Agent$ActiveLifeCycle.transitionTo(...) (0.00 ns)
 ——————————— 0.0% jade.core.Agent.notifyChangedAgentState(...) (10.00 ms)
 ——————————— 0.0% java.lang.Thread.currentThread(...) (0.00 ns)
 ——————————— 0.0% java.lang.Object.equals(...) (0.00 ns)
 ——————— 0.0% java.lang.Thread.sleep(...) (0.00 ns)
 ————— 0.0% jade.core.Agent.notifyChangeBehaviourState(...) (0.00 ns)
 ———— 0.0% jade.core.behaviours.CyclicBehaviour.done(...) (0.00 ns)
```

(a)

Jade Agent Administration
jade.core.Agent.getAMS
Jade Mobility
jade.core.Agent.doMove
jade.core.mobility.AgentMobilityService.TransitLifeCycle
jade.core.Agent.ActiveLifeCycle.transitionTo
Jade Directory Services
jade.core.Agent.notifyChangedAgentState

(b)

Fig. 4. (a) Runtime trace that drives the process view. (b) Jade mobility process view activity diagram.

diagram exists (Fig. 4). The darkened circle is the starting point when the feature slice is executed. Activities are ovals depicting method invocations. Arrows represent flow between activities and packages encompassing the activities represent functional concepts. The resulting concrete architectures evoke the abstract process architecture by combining common activities and descriptions where and why activities may differ across agent framework implementations.

Our software analysis showed subtle distinctions in how these representative frameworks instantiated the mobility concept. In Jade, when an agent moves to another container, it invokes its own doMove method, supplying the target location

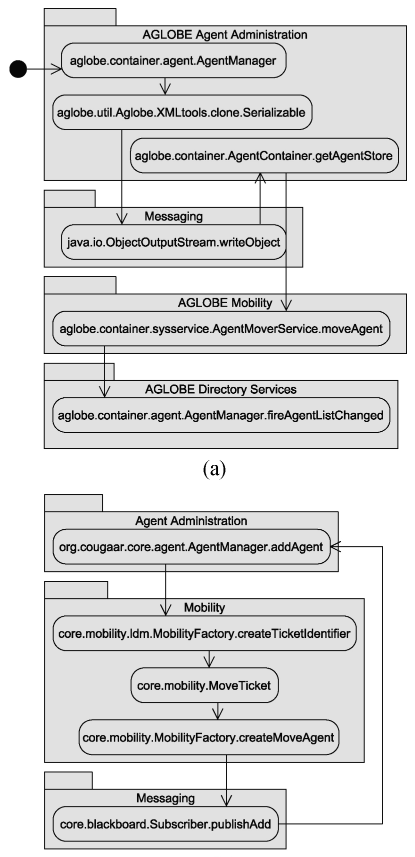

AGLOBE Agent Administration
aglobe.container.agent.AgentManager
aglobe.util.Aglobe.XMLtools.clone.Serializable
aglobe.container.AgentContainer.getAgentStore
Messaging
java.io.ObjectOutputStream.writeObject
AGLOBE Mobility
aglobe.container.sysservice.AgentMoverService.moveAgent
AGLOBE Directory Services
aglobe.container.agent.AgentManager.fireAgentListChanged
(a)
Agent Administration
org.cougaar.core.agent.AgentManager.addAgent
Mobility
core.mobility.ldm.MobilityFactory.createTicketIdentifier
core.mobility.MoveTicket
core.mobility.MobilityFactory.createMoveAgent
Messaging
core.blackboard.Subscriber.publishAdd
(b)

Fig. 5. AGLOBE and Cougaar UML activity diagrams. (a) AGLOBE mobility process view. (b) Cougaar mobility process view.

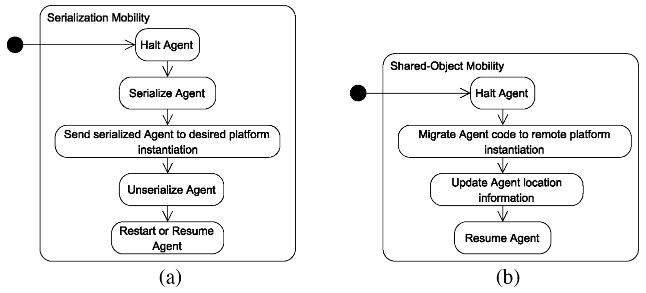

Serialization Mobility
Halt Agent
Serialize Agent
Send serialized Agent to desired platform instantiation
Unserialize Agent
Restart or Resume Agent
(a)
Shared-Object Mobility
Halt Agent
Migrate Agent code to remote platform instantiation
Update Agent location information
Resume Agent
(b)

Fig. 6. Process view: mobility functional concept. (a) Serialization architecture. (b) Shared object architecture.

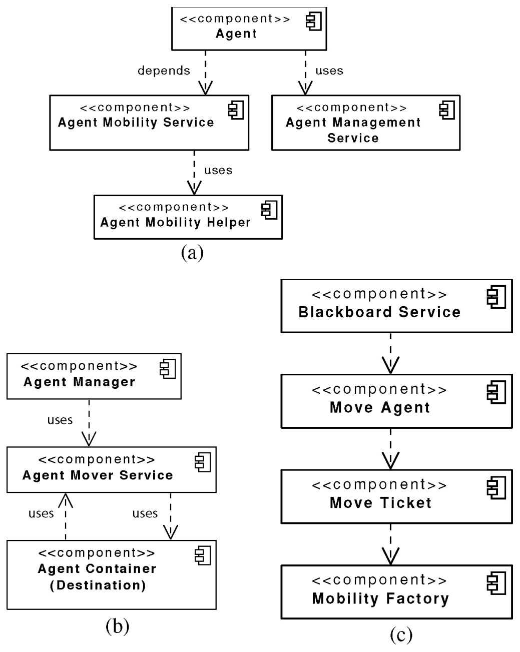
as an argument to the agent management service (AMS). This target location container is obtained by querying the AMS for available containers in the agent system or by querying the AMS for a particular agent's location. Invoking `doMove` changes the state of the agent for transmission, pausing its execution and allowing the Agent Mobility Service to move the agent. The Jade container serializes the agent and sends it to the destination container (while removing it from the original container). The destination container deserializes the object, yielding a copy of the agent; this new agent assumes the name and identity of the old agent. When the move is complete, the agent mobility service informs the agent, and the agent's state changes from paused to active. The agent's execution continues. Fig. 4 provides a temporal view of the scenario showing method invocations and package structures. In constructing the ASRA, one must choose method invocations that highlight the mobility functional concept and interactions with agent administration functional concepts, resulting in the activity diagram (Fig. 4).

AGLOBE's approach is simpler, using an AgentMover service that creates a clone of the current agent and moves it to the specified container. AGLOBE uses Java's Serialization API and ObjectOutputStream to serialize and transmit the agent information to the new container. If there is only a single agent in the original container, AGLOBE removes the original container. Lastly, Cougaar initializes mobile agents to inform the framework of their mobile status and their current location. Copies of dormant mobile agents are initialized on every remote container in the system. During execution, the state of the agent is maintained on each platform via the Cougaar blackboard. When an agent moves to another platform, it creates a ticket with the new container location and publishes this to the blackboard. The new container retrieves the ticket, initializes the dormant agent, and resumes execution. Once the dormant agent is started, it sends an acknowledgment back to the sending node so the original copy can be stopped. Fig. 5 presents the UML activity diagrams for AGLOBE and Cougaar mobility.

Resulting mobility process view patterns: Comparing the diagrams produced by analysis, two patterns for agent mobility emerge. Jade and AGLOBE exhibit serialization mobility (Fig. 6) in which an agent's execution is paused, converted into a transferable form, transmitted to a target platform, converted into an executable form, and resuming agent's execution. Cougaar exhibits shared-object mobility (Fig. 6) in which agents exist on all platform containers and the agent's state is synchronized across platforms during execution. Agent mobility is achieved by changing the shared state to the new platform location.

3) *Creating the Implementation View:* The implementation view is the static view of the agent framework derived through static analysis of source code and identifying components and activities from the process view. As such, this correlates to the actual software architecture approach to implementation of the agent system. UML component diagrams in the Implementation View represents high-level components rather than objects. Jade, AGLOBE, and Cougaar mobility concrete view diagrams are shown in Fig. 7.

Agent mobility in Jade consists of four software components shown in Fig. 7. The agent component depends and uses the agent mobility service and the agent management service. The agent mobility service converts the mobile agent into a transferable form (i.e., serialize the agent) using the agent mobility helper component. The agent management service pauses the agent execution and locates the destination container. In AGLOBE, mobility consists of three components shown in Fig. 7. The agent manager locates the destination platform, pauses the agent, serializes the agent's state, and passes the agent information to the container service. The mover service handles communication and transport with the destination container component. Lastly, Cougaar mobility consists of



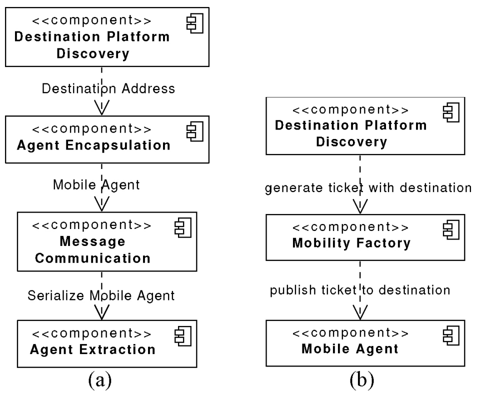
<<component>> Agent
depends
uses
<<component>> Agent Mobility Service
<<component>> Agent Management Service
uses
<<component>> Agent Mobility Helper
(a)
<<component>> Agent Manager
uses
<<component>> Agent Mover Service
uses
uses
<<component>> Agent Container (Destination)
(b)
<<component>> Blackboard Service
<<component>> Move Agent
<<component>> Move Ticket
<<component>> Mobility Factory
(c)

Fig. 7. Implementation view of (a) Jade, (b) AGLOBE, and (c) Cougaar UML package diagrams for mobility.

four components shown in Fig. 7. The MobilityFactory, in the Cougaar core package, generates a ticket identifier. The ticket can be created directly by invoking its constructor using this identifier. Using this ticket, a MoveAgent object can be created through the MobilityFactory. The MoveAgent is a unique object identifier that represents the state of the agent to be moved. When the MoveAgent is published to the blackboard, the agent state is serialized and copied to the receiving agent container.

Resulting mobility implementation view patterns: Based on the software analysis results, there appear two canonical patterns for mobility's implementation view (Fig. 8): serialization mobility and ticketing mobility. Jade and AGLOBE mobility follow a serialization mobility pattern (Fig. 8). The platform discovery component uses directory services to find the destination platform. The agent encapsulation component creates a representation of the mobile agent for transport. The messaging component delivers the mobile agent to the destination platform. Finally, the agent extraction component receives the mobile agent, loads it in the platform, and resumes its execution. In contrast, the Cougaar mobility follows a ticketing system pattern (Fig. 8). The platform discovery component uses the directory services component to find the destination platform. A mobility factory component generates a ticket ID to identify the destination platform of the mobile agent. Finally, the mobile agent component uses messaging functional concept to publish the ticket to the other hosts.

4) *Construct the Logical View for Mobility:* The logical view expresses the high level component interactions among functional concepts in an agent system. While software designers and architects often use logical models to describe intended system behaviors, the actual interactions present in an implementation can often differ from those in an idealized or abstract architecture. The logical view is constructed by observing the clustered runtime data generated from EJP and BAT and organizing the major objects into packages. This organization is represented with UML package diagrams (Fig. 9) for the Jade, AGLOBE, and Cougaar mobility functional concept. Mobility in Jade utilizes the agent administration components (Fig. 9). The AMS package handles identifying target platforms. The agent package depends on the AMS package and is responsible for moving the agent to the target platform.



<<component>> Destination Platform Discovery
Destination Address
<<component>> Agent Encapsulation
Mobile Agent
<<component>> Message Communication
Serialize Mobile Agent
<<component>> Agent Extraction
(a)
<<component>> Destination Platform Discovery
generate ticket with destination
<<component>> Mobility Factory
publish ticket to destination
<<component>> Mobile Agent
(b)

Fig. 8. Mobility UML diagrams. Implementation view of (a) serialization mobility and (b) ticketing mobility.

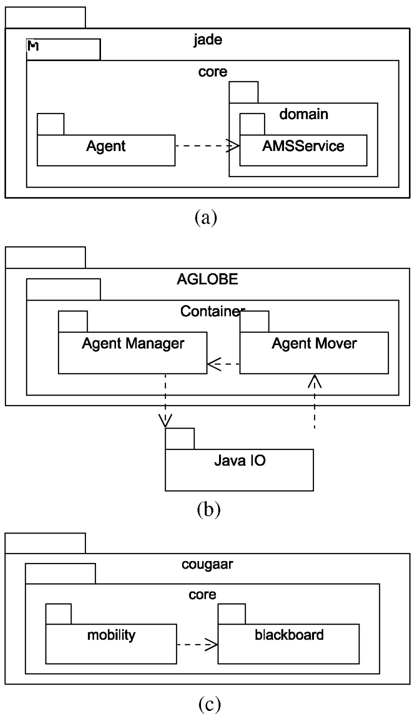


Fig. 9. Logical view of (a) Jade, (b) AGLOBE, and (c) Cougaar mobility UML package diagrams.

AGLOBE's mobility logical view consists of three packages: 1) agent manager; 2) agent mover; and 3) Java IO. When an agent migrates to another container, it first uses the agent manager package to initiate the procedure. Then, the agent manager contacts the Java IO library to serialize and send the agent to the specified container. Next, the agent mover package

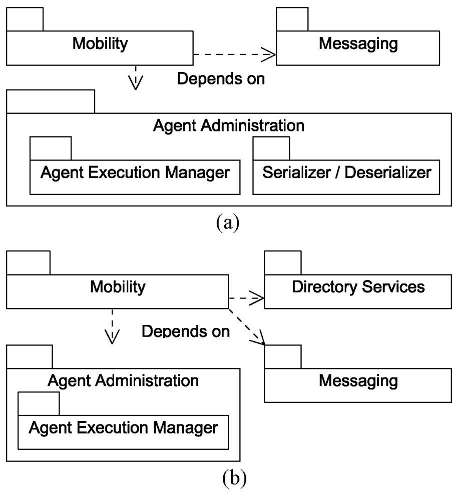


Fig. 10. Logical view: two paradigms for mobility. (a) Serialization logical view: migration component depends on the agent execution manager and serialization components of the agent controller component and the messaging component. (b) Shared object paradigm: migration component depends on directory services component, agent controller component, and messaging component.

alerts the target container of the migrating agent. Finally, the agent manager updates the target container with the migrated agent and removes the agent from the source container as seen in Fig. 9. In contrast, Cougaar's mobility features allow agents to move between different nodes (computers or Java virtual machine instances) during execution. The package decomposition of Cougaar mobility in Fig. 9 illustrates the usage of the Mobility and Blackboard components in the core packages of Cougaar.

Resulting mobility logical view patterns: The logical view for mobility depicts two patterns: serialization mobility and shared object mobility. Jade and AGLOBE follow the serialization mobility pattern in which the agent is converted to a transferable form before migrating the agent to its destination. The mobility functionality (Fig. 10) depends on the agent administration to pause and start the agent and messaging components to transmit the agent. Cougaar follows the shared object mobility pattern in which the agent representation is shared among platforms. Agent mobility involves synchronizing the state of the agent then halting the agent on the source platform and initializing and executing the agent on the target instance. Shared object mobility (Fig. 10) depends on the agent administration component for halting and initializing the agents, the messaging component for synchronizing the state, and directory services for finding the target platform.

IV. Summary of Resulting Architecture

This section summarizes the abstract architectural descriptions of the six other functional concepts for agent systems, the complete details of which are available in [39].[3]

Messaging functional concept: The messaging functional concept specifies how agents communicate within an

[3]The complete document is available at http://edge.cs.drexel.edu/people/regli/ACIN-reference_architecture_v1.0.pdf.

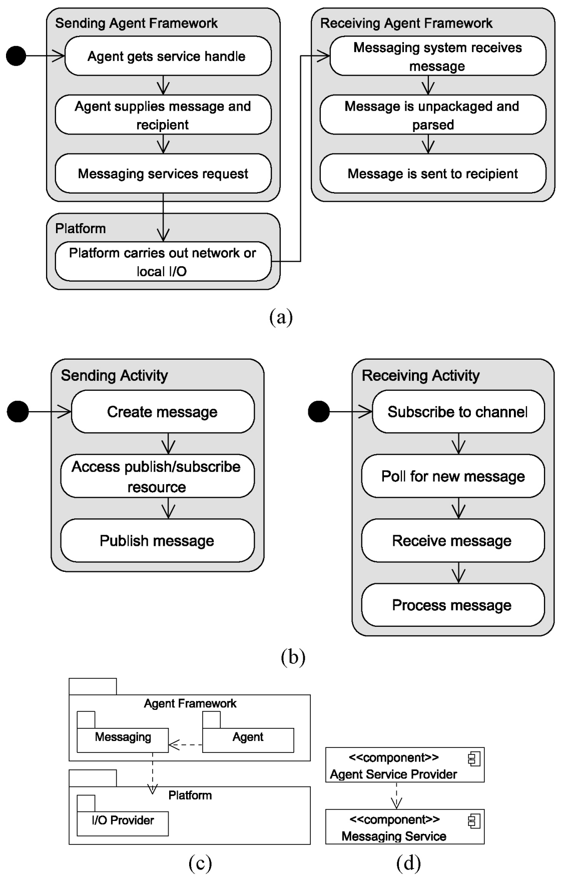

Sending Agent Framework
Agent gets service handle
Agent supplies message and recipient
Messaging services request
Receiving Agent Framework
Messaging system receives message
Message is unpackaged and parsed
Message is sent to recipient
Platform
Platform carries out network or local I/O
(a)
Sending Activity
Create message
Access publish/subscribe resource
Publish message
Receiving Activity
Subscribe to channel
Poll for new message
Receive message
Process message
(b)
Agent Framework
Messaging
Agent
Platform
I/O Provider
<<component>> Agent Service Provider
<<component>> Messaging Service
(c)
(d)

Fig. 11. Messaging architectural views. (a) Direct messaging process view. (b) Reference messaging process view. There are separate activities occurring asynchronously for the sender and receiver. (c) Logical view. (d) Implementation view.

agent-based system. The ASRM states, the messaging functional concept involves a source, a channel, and a message. Messaging does not necessarily involve recipient(s). For instance, an agent broadcasting a message or modifying the surrounding environment are examples of messaging without designated receivers. There are many scenarios for the messaging functional concept.

In the scenario view, the source agent first creates the message and appends the destination agent's address by retrieving it from the directory services component based upon the destination agent's name. The agent framework platform's message transport system delivers the message to the destination agent. The destination platform receives the message and delivers it to the receiving agent. The message serialization step encodes the message to be named, packaged, and delivered in some manner and awaits message retrieval, in which messages may be processed synchronously or asynchronously, either polling the queue at repeated intervals or by handling events.

The process view for messaging consists of two abstractions: direct messaging and reference messaging. With direct messaging (Fig. 11), agents directly send a message to another agent (or agents). The architecture for direct messaging requires seven steps.

1) The source agent creates the message.

2) The source agent adds the destination agent's address to the recipient field of the message by using the directory services component.
3) The source agent adds the content data to the message.
4) A function/method passes the message to the source agent's local container.
5) The source agent's local container sends the message to the destination agent's container.
6) The destination agent's container receives the message and passes it to the destination agent.
7) A function/method of the destination agent is called and the destination agent reads the message.

In contrast, reference messaging is an indirect method of messaging in which three meta messages must be sent between source and destination agents for one message to be sent. However, this method allows asynchronous messaging since destination agent(s) retrieve messages from a single location (i.e., a publish/subscribe channel). Reference messaging (Fig. 11) consists of six steps.

1) The source agent creates the message.
2) The source agent adds the destination agent's address to the recipient field of the message by using the directory services component.
3) The source agent adds the content data to the message.
4) A function/method publishes the message on the source agent's local data store and then sends a reference of that location to the destination agent.
5) The destination agent receives the reference, subscribes to the source agent's data store, and retrieves the message from the source agent's data source.
6) A function/method is called and the destination agent reads the message.

Although this view only describes one-to-one messages, agents can also send one-to-many messages. One-to-many messaging consists of two abstractions: multiple one-to-one messages and multicast. Multiple one-to-one messages is when an agent sends a single message to each agent individually using container or reference messaging. Multiple one-to-one messaging is preferred if a reliable connection is required and messaging overhead is not a factor. Multicast messaging is when a framework utilizes the underlying platform to send out a multicast message to the agents within that group. From the implementation view, messaging consists of a single abstraction (Fig. 11) with two components: an agent service provider responsible for providing the messaging functionality, and the messaging service provider to handle transportation of the message for the agent platform. The agent framework makes these components available as interfaces to agent developers. Subsequently, the logical view is a single abstraction (Fig. 11) in which the agent package encapsulates the messaging functionality for the agent the message sending. The agent package depends on the container package to find the destination address of the recipient.

Agent administration functional concept: The agent administration functional concept gives a notion of command and control to agent frameworks to create, manage, and terminate agents. This functional concept may depend on other functional concepts such as directory services and security used to identify specific agents or agent populations. Security is sometimes required to keep unprivileged agents from being instantiated, modified, or terminated. A logging module may be required to capture the agent administration tasks. Fig. 12 shows the agent administration functional concept views.

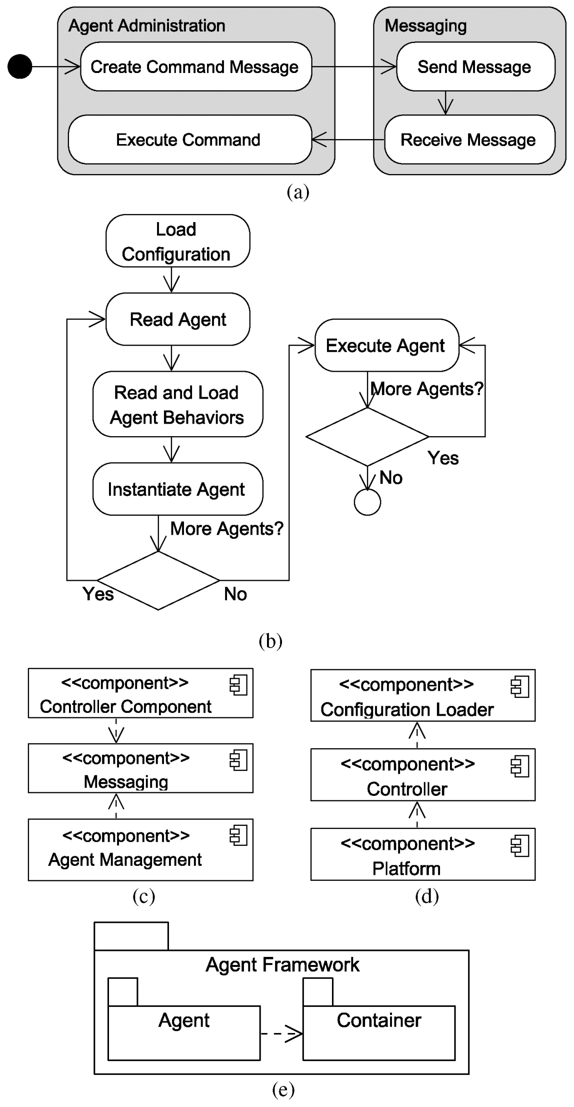


Fig. 12. Agent administration architectural views. (a) Process view: dynamic agent creation. (b) Process view: static agent creation. (c) Implementation view: dynamic agent creation. (d) Implementation view: static agent creation. (e) Logical view.

The scenario view for agent administration is defined by three use cases: creating agents, managing agents, and terminating agents. In create agent the framework or another agent instantiates a new agent. Many other scenarios depend on this use case, i.e., to name an agent as in directory services, an agent must first be spawned. Similarly, the process of moving an agent across platforms, as seen in the mobility functional concept, also necessitates prior instantiation of an agent. The manage agent use case enables prioritization of resource usage, particularly, in environments when those resources are constrained. Some frameworks offer functionality to control how frequently or how intensely agents may access resources.

In some cases, the delegation of resource usage is left to the underlying platform (i.e., the operating system). Lastly, terminate agent provides several mechanisms that terminate agents: an agent can be terminated by itself, another agent, or an external entity.

To illustrate the process view, this paper provides one representative use case for dynamically or statically creating and instantiating a new agent in an agent system. In dynamic agent creation, administration API enable runtime viewing and managing agents in an overall agent system. Fig. 12 shows the activity diagram in which commands are issued through the administrative interface, sent via the messaging component, and commands performed on the agent platform to instantiate the agent. In static agent creation agents can be instantiated via configuration files that specify the agent's behaviors. This approach can enable system initialization with complex communities, groups, and societies of agents *a priori*. Fig. 12 depicts the static agent creation process.

The implementation view for agent administration shows two UML component diagrams describing the high-level components and their dependencies. Dynamic agent creation (Fig. 12) is accomplished through issuing commands to an agent. The commands are issued as messages from the controller component, delivered via the messaging component, and consumed by the agent management component. This pattern is followed for dynamic agent creation and consists of three primary components. The controller depends on the messaging component and the agent management component depends on the messaging component. In static agent creation (Fig. 12) an initializer component to read the configuration files, a controller component to instantiate the agents and their behaviors, and the platform component to hold the instantiated agents. The controller depends on the initializer and the platform depends on the controller.

Lastly, the logical view (Fig. 12) using a container package to issue a command, and the agent package to receive and execute the command. All agent frameworks examined had similar package dependency structures for administration.

Directory services functional concept: The directory services functionality facilitates locating and accessing shared resources. Fig. 13 shows the abstract architectural process, implementation, and logical view diagrams. From the scenario view, the primary use cases for directory services are publishing, advertising, removing, and querying of services accessible to an agent. Processes within an agent framework need a way to reference services. This is accomplished via a naming scheme and a directory maintained by the framework holding these service names, locations, descriptions, and metadata. When a service appears in the registry, it is available for querying. When a service is no longer available, it must be removed from the directory. The process view defines a subscription-based process for directory services that provide naming, notification, and query matching processes, illustrated in the UML process diagram in Fig. 13 as having three key steps: 1) an agent contacts the directory services manager to request a resource; 2) the directory services manager notifies the other agents subscribed to the resource; 3) agents deliver the resources to the requesting agent. To illustrate the implementation view, a directory services element consists of four components (Fig. 13). A request producer may be an agent or other service. The directory services manager services the requests by decoding, interpreting, and/or reasoning over the query before sending to the Request Relay. The request relay determines and transfers the request to the appropriate request executor and the request executor executes the request. Lastly, the logical view (Fig. 13) focuses on three key packages in the agent framework, the agent definition, and messaging. The package containing the agent components use the Communicator package to send a message to the directory service manager package.

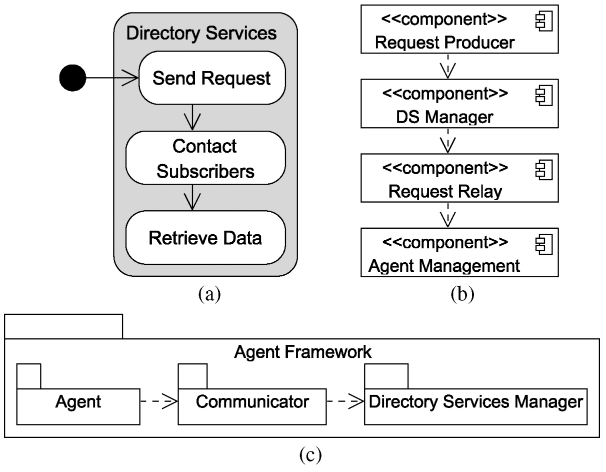


Fig. 13. Directory services architectural views. (a) Process view. (b) Implementation view. (c) Logical view.

Conflict management functional concept: The conflict management functional concept (Fig. 14) is an agent framework's systems to perform classification, avoidance, detection, negotiation, and resolution of conflicts among agents. Conflict management and coordination represent central elements of distributed, autonomous agent-based systems, and hence this functional concept was found as a common component of the reference architecture. The literature in this area is quite diverse and an in-depth discussion of various approaches that might be included here is beyond the scope of this study. Interested readers are referred to recent literature covering topics in negotiation [40], conflict resolution [41], [42], and multiagent coordination [43].

What is discovered in reviewing the existing frameworks and related literature is that the approaches to conflict and resolution are nearly always dependent on the application domain. Hence, the presentation of this functional concept is considerably more abstract than that of other agent framework components.

From the scenario view, conflicts in multiagent systems occur when agents vie for a limited environmental resource. At an abstract level, such agent concepts such as goals, BDI, etc. can also be viewed as instances resources.[4] Conflicts can be viewed as occurring in two categories: goal and plan conflicts. A goal conflict occurs when the function or purpose of an

[4]This distinction is admittedly arbitrary, i.e., one could do the reverse and represent resources and their consumption as goal or BDI concepts.

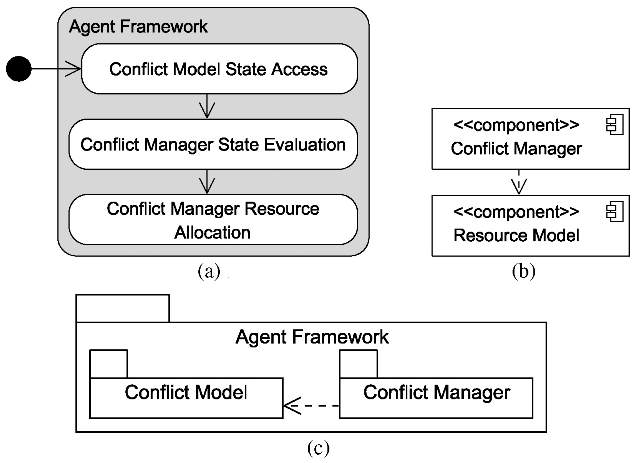

Agent Framework
Conflict Model State Access
Conflict Manager State Evaluation
Conflict Manager Resource Allocation
(a)
<<component>> Conflict Manager
<<component>> Resource Model
(b)
Agent Framework
Conflict Model
Conflict Manager
(c)

Fig. 14. Conflict management architectural views. (a) Process view. (b) Implementation view. (c) Logical view.

agent (or group of agents) directly opposes the purpose or goal of another agent (or group of agents). Plan conflicts result from scheduling resources at the same time, or any post-condition of an agent conflicting with the precondition of another agent. Each conflict category is detected and address through different techniques that raise issues and problems.

There are many methods for managing conflicts. Conflict avoidance allows agents to execute their plans until a conflict occurs and deal with the conflict through appropriate actions, conflict can also be avoided. By analyzing and monitoring agent goals and plans before their execution, one can determine whether conflicts may exist. This can be accomplished through planning algorithms or entities that monitor agent behavior to ensure that conflicts do not occur. Conflict detection occurs by an entity observing the system to detect conflicts in goals, actions, or resources. When a conflict is detected by this entity, it is sent to the conflict resolution module. While the instantiation of these architectural elements will be highly domain dependent, the subject of conflict management in agent systems is a well-studied problem. Kakehi [44] proposes directly communicating agents' plans to one another when they think they may conflict. Another method, proposed by Sugawara [45], utilizes an agent manager to analyze agent plans in determining whether a conflict may occur.

Conflict negotiation, which differs from avoidance, is the one-on-one interaction of agents to resolve a conflict. While this is used in some systems, it is not universal to all methods of conflict management. An example from the literature is Lopez [46], which generalizes a structure for agent negotiation to resolve one-on-one conflicts between agents. Lastly, conflict resolution fixes unavoidable conflicts that occur in the system by altering the system or agents' state to resolve the conflict while preserving the integrity of the agents and system execution. If agents' goals conflict, the goals are redefined or constraints and conditions are relaxed. An example of this case is the work of Oliveira [47], that incorporates the use of a cooperation layer that is assigned agent tasks.

The process view of conflict management features (Fig. 14) in an agent framework must support some internal model of resources, which agents can collaboratively manage. Other agents, responsible for conflict management, adjust the state of the resource model to remove the conflict, as defined by the model. From the process view, since there is no particular requirement about the model of resources and conflicts except that it exists and that agents resolve conflicts, the process view depends entirely on the model and the user-implemented means of conflict management. The system is only responsible for giving the user the ability to construct the model and agents responsible for conflict management. The implementation view must include structures (e.g., classes) that allow system designers to define a model of resources. It must also contain an interface that allows running agents to monitor and manipulate these resources (Fig. 14). The framework should provide sufficient resources for dynamic planning applications. Lastly, the logical view (Fig. 14), like the process view, is specific to the agent application the user wishes to build.

Security and survivability functional concept: From a scenario view (Fig. 15), the security and survivability functional concept covers an agent framework's ability to perform authentication, authorization, and enforcement. Authentication is accomplished by verifying the source of an agent. Authorization is the process by which an agent framework decides if an agent has the privilege to perform a task. Authorization depends on the authentication of the agent. Enforcement is the mechanism to permit a trusted resource to execute a task. A framework utilizes authorization to grant/deny access to an agent's request. In many cases, some of these capabilities may be handled by the underlying platform (OS or runtime environments on hosts), hence a full discussion of how to provide security features involves many domain-specific and implementation-specific details and is thus is beyond the scope of the ASRA.

From a process view (Figs. 15 and 15), when an agent sends some kind of secured or private message it must acquire and sign some kind of security certificate. The receiving agent verifies the certificate and then begins to send trusted messages. This also affects agent migration. The implementation view (Fig. 15) notes that every security function consists of a service, proxy, and implementation component, following a proxy design pattern. Lastly, the logical view can be architecturally broken into one or more modules providing these features. For instance, if authentication is supported, there should be a module that allows the framework to create users, allow them to supply credentials, and authenticate them for the agent application. From there, another module may be used to encrypt communications using keys that belong to users on the system, and another might be used to provide enforcement, disallowing users from performing actions that they are not permitted to perform.

Logging functional concept: Lastly, the logging functional concept covers an agent framework's ability to perform log generation, storage, and access. Designers specify system events that triggers log messages to be recorded corresponding to that event. These events can correspond to actions in any functional area and with any action an agent takes. The process view produced by runtime analysis of the agent frameworks under review shows that when an event is logged, the agent

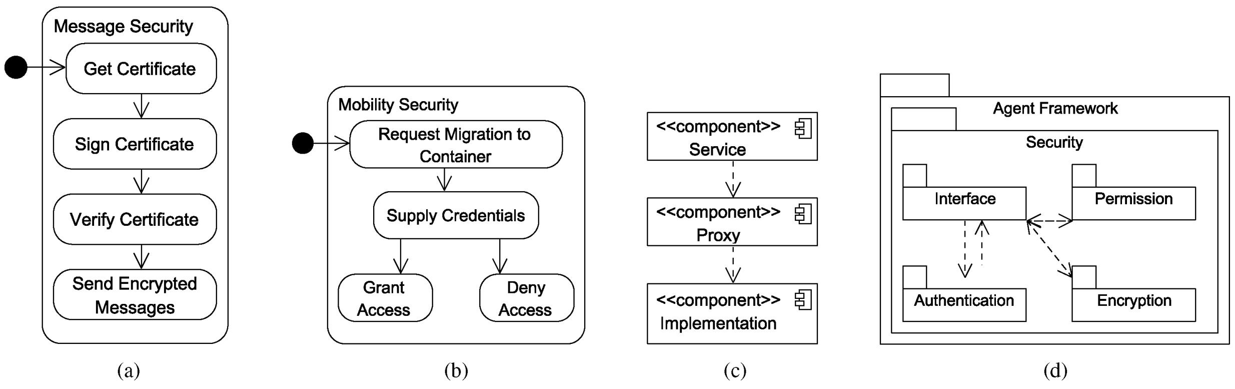


Fig. 15. Security architectural views. (a) Process view: message security. (b) Process view: mobility security. (c) Development view for security. (d) Logical view for security.

framework exposes logging through a facade pattern to an underlying logging facility. When an agent logs an event, the agent makes a call to the agent framework log wrapper, which passes the call to the underlying log facility. From an implementation view, the process of logging an event passes through two components, the logging service and the underlying logging facility. The logical view for these systems revealed that the agent frameworks were utilizing the underlying platform implementations for logging.

V. Using the ASRA

The ASRA is an elaboration of the ASRM [17], [18]. The goal has been to further establish relationships between the ASRM functional concepts in agent frameworks and define the architectural patterns for these concepts. As a reference architecture, the ASRA does not address design or implementation specifics—focusing on the possible interactions between functional concepts.

For the ASRA, stakeholders are agent system architects and developers, (i.e., those building systems-of-systems that include agent technology) the individuals involved in creating designing agent frameworks and agent systems and implementing agent systems. Software architects and developers are concerned with individual agent behaviors and how they contribute to overall system behavior. Additionally, these stakeholders need to concern themselves with implementation or selection of agent framework architectures. Agent system architects are concerned with the high-level purpose and scope of functional concepts, their interaction with other functional concepts and their greater involvement in an agent-based system. Conversely, agent system developers are concerned with the functionality of each concept's implementation details and how each concept contributes to the capabilities of the overall agent system. The ASRA provides value to these stakeholders including the following.

1) *Agent system developers:* The ASRA helps system developers by explaining the abstract features of agent systems, thus documenting the capabilities of a particular agent framework at a high-level.
2) *Agent system designers:* The ASRA helps agent system designers explain how agents' functional concepts can be best integrated for a specific application. For example, one might choose Cougaar's agent framework for its Blackboard messaging functionality rather than AGLOBE's more conventional Direct messaging design.
3) *Agent framework designers:* The ASRA illustrates best practices for implementing functional concepts, thus, helping agent framework designers decide how to best design their frameworks.
4) *Standards developers:* The ASRA provides interconnection blueprints between standardized functional concepts and layers from a reference model.

In the course of this paper, the authors used the ASRA to develop an overall architecture for system for network-centric command and control. This underlying design and implementation for the tactical information technologies for assured network operations (TITAN) program [48], [49] was based on heterogeneous agents and web services that provided information management and dissemination services in support of military planning and operations. The ASRA provided the overarching methodology for making architectural decisions as to how to design and implement various functional concepts at the overall, systems-level. In some functional areas, decisions were left to the scope of agent providers (i.e., agents were responsible for managing their own resources and conflicts); in others, shared functionality was provided (e.g., a common message infrastructure).

VI. Conclusion

This paper presented a process, based on forensic software analysis, for creating reference architectures and demonstrated the application of this process to create an agent systems reference architecture. The process used software analysis techniques and presented the architecture as a set of views that address the analytic and developmental needs of architects, developers, and managers. The basis of the ASRA were the functional concept definitions of the ASRM [17], [18] and was presented as a set of architectural views for each functional concept. The resulting ASRA was a reference architecture for systems composed of agents and created by current agent software frameworks. This stance does not prescribe how one should create an agent system, but rather, allows architects, designers, and developers to make informed decisions on which architectural components are necessary for their particular system needs. It was the authors' belief that the ASRA, coupled with standards from FIPA and others, provided a basis for improving software engineering practice for heterogeneous agent-based systems. In addition, the ASRA will facilitate adoption of agent system development processes by providing blueprints for constructing agent systems and improving their interoperability.

References

[1] N. R. Jennings, "An agent-based approach for building complex software systems," *Commun. ACM*, vol. 44, no. 4, pp. 35–41, 2001.

[2] A. Helsinger, M. Thomas, and T. Wright, "Cougaar: A scalable distributed multi-agent architecture," in *Proc. IEEE Int. Conf. Syst., Man, Cybern.*, Oct. 2004, pp. 1910–1917.

[3] M. Pěchouček and D. Šišlák, "Agent-based approach to free-flight planning, control, and simulation," *IEEE Intell. Syst.*, vol. 24, no. 1, pp. 14–17, Jan.–Feb. 2009.

[4] D. Weyns, H. V. D. Parunak, and O. Shehory, "The future of software engineering and multi-agent systems," *Int. J. Agent-Oriented Softw. Eng.*, vol. 3, no. 4, pp. 369–377, 2009.

[5] The Foundation for Intelligent Physical Agents. (2002) [Online]. Available: http://www.fipa.org

[6] Foundation for Intelligent Physical Agents. (2002, Dec.). *Abstract architecture* [Online]. Available: http://www.fipa.org/specs/fipa00001/

[7] Foundation for Intelligent Physical Agents. (2002, Mar.). *FIPA agent management specification* [Online]. Available: http://www.fipa.org/specs/fipa00023/

[8] P. Kruchten, "Architectural blueprints: The '4+1' view model of software architecture," *IEEE Softw.*, vol. 12, no. 6, pp. 42–50, Nov. 1995.

[9] P. Kruchten, *The Rational Unified Process: An Introduction*, 3rd ed. Reading, MA, USA: Addison-Wesley Professional, 2003.

[10] M. Khalgui and H. Hanisch, "Reconfiguration protocol for multi-agent control software architectures," *IEEE Trans. Syst., Man, Cybern. C, Appl. Rev.*, vol. 41, no. 1, pp. 70–80, Jan. 2011.

[11] J. Vokřínek, A. Komenda, and M. Pěchouček, "Abstract architecture for task-oriented multi-agent problem solving," *IEEE Trans. Syst., Man, Cybern. C, Appl. Rev.*, vol. 41, no. 1, pp. 31–40, Jan. 2011.

[12] K. T. Seow, K. M. Sim, and S. Kwek, "Coalition formation for resource coallocation using BDI assignment agents," *IEEE Trans. Syst., Man, Cybern. C, Appl. Rev.*, vol. 37, no. 4, pp. 682–693, Jul. 2007.

[13] A. Sturm, D. Dori, and O. Shehory, "An object-process-based modeling language for multiagent systems," *IEEE Trans. Syst., Man, Cybern. C, Appl. Rev.*, vol. 40, no. 2, pp. 227–241, Mar. 2010.

[14] L. Cao, C. Zhang, and M. Zhou, "Engineering open complex agent systems: A case study," *IEEE Trans. Syst., Man, Cybern. C, Appl. Rev.*, vol. 38, no. 4, pp. 483–496, Jul. 2008.

[15] C. Bernon, M. Cossentino, M. Pierre Gleizes, P. Turci, and F. Zambonelli, "A study of some multi-agent meta-models," in *Proc. 5th Int. Workshop Agent-Oriented Softw. Eng.*, 2004, pp. 62–77.

[16] G. Beydoun, G. Low, B. Henderson-Sellers, H. Mouratidis, J. Gomez-Sanz, J. Pavon, and C. Gonzalez-Perez, "FAML: A generic metamodel for MAS development," *IEEE Trans. Softw. Eng.*, vol. 35, no. 6, pp. 841–863, Nov.–Dec. 2009.

[17] W. C. Regli, I. Mayk, C. J. Dugan, J. B. Kopena, R. N. Lass, P. J. Modi, W. M. Mongan, J. K. Salvage, and E. A. Sultanik, "Development and specification of a reference model for agent-based systems," *IEEE Trans. Syst., Man, Cybern. C*, vol. 39, no. 5, pp. 572–596, Sep. 2009.

[18] W. C. Regli, I. Mayk, C. J. Dugan, J. B. Kopena, R. N. Lass, P. J. Modi, W. M. Mongan, J. K. Salvage, and E. A. Sultanik. (2006, Nov.). "Agent systems reference model," Applied Communications and Information Networking Program, U.S. Army CERDEC and Drexel University, Camden, NJ, USA, Tech. Rep. V1a [Online]. Available: http://www.fipa.org/docs/ACIN-reference_model-v1a.pdf.

[19] K. Laskey, J. A. Estefan, F. G. McCabe, and D. Thornton. (2009). "Reference architecture foundation for service oriented architecture," OASIS, Tech. Rep. [Online]. Available: http://docs.oasis-open.org/soa-rm/soa-ra/v1.0/soa-ra.html.

[20] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*. Reading, MA, USA: Addison-Wesley Professional, 2003.

[21] M. Fowler and K. Scott, *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, 2nd ed., Boston, MA, USA: Addison-Wesley Longman, 2000.

[22] Y. Zhou, Y. Chen, and H. Lu, "UML-based systems integration modeling technique for the design and development of intelligent transportation management system," in *Proc. IEEE Int. Conf. Syst. Man Cybern.*, Sep. 2004, pp. 6061–6066.

[23] H. Zimmerman, "OSI reference model—The ISO model of architecture for open system interconnection," *IEEE Trans. Commun.*, vol. COM-28, no. 4, pp. 425–432, Apr. 1980.

[24] ANSI/IEEE. (2009). *Recommended practice for architectural description of software-intensive systems* [Online]. Available: http://www.iso-architecture.org/ieee-1471

[25] P. Clements, D. Garlan, L. Bass, J. Stafford, R. Nord, J. Ivers, and R. Little, *Documenting Software Architectures: Views and Beyond*. Reading, MA, USA: Addison-Wesley Professional, 2002.

[26] M. Collins-Cope and H. Matthews, "A reference architecture for component based development," in *Proc. 6th Int. Conf. Object Oriented Inform. Syst.*, 2000, pp. 225–237.

[27] D. Weyns and T. Holvoet, "A reference architecture for situated multi-agent systems," in *Proc. 3rd Int. Conf. Environ. Multi-Agent Syst. III* , vol. 4389, 2006, pp. 1–40.

[28] N. Cai, M. Gholami, L. Yang, and R. W. Brennan, "Application-oriented intelligent middleware for distributed sensing and control," *IEEE Trans. Syst., Man, Cybern. C, Appl. Rev.*, vol. 42, no. 6, pp. 947–956, Nov. 2012.

[29] P. Avgeriou, "Describing, instantiating, and evaluating a reference architecture: A case study," *Enterprise Architect J.*, San Mateo, CA, USA: Fawcette Tech., Jun. 2003.

[30] W. Eixelsberger, "Recovery of a reference architecture: A case study," in *Proc. Third Int. Workshop Softw. Architecture*, 1998, pp. 33–36.

[31] J.-M. DeBaud, O. Flege, and P. Knauber, "PuLSE-DSSA—A method for the development of software reference architectures," in *Proc. 3rd Int. Workshop Softw. Architecture*, 1998, pp. 25–28.

[32] J. S. Albus, "Outline for a theory of intelligence," *IEEE Trans. Syst., Man, Cybern.*, vol. 21, no. 3, pp. 473–509, May–Jun. 1991.

[33] J. A. Albus, H. McCain, and R. Lumia, "NASA/NBS standard reference model for telerobot control system architecture (NASREM)," Natl. Inst. Standards Technol., Tech. Rep. 1235, 1989.

[34] W. M. Mongan, C. J. Dugan, R. N. Lass, A. K. Hight, J. Salvage, W. C. Regli, and P. J. Modi, "Dynamic analysis of agent frameworks in support of a multiagent systems reference model," in *Proc. Int. Conf. Intell. Syst. Agents (Part MCCSIS)*, 2007, pp. 11–18.

[35] F. Bellifemine, G. Caire, A. Poggi, and G. Rimassa, "Jade: A white paper," Telecom Lab., TILAB, Torino, Italy, Tech. Rep. 3, Sep. 2003.

[36] D. Šišlák, M. Rehák, M. Pěchouček, and D. Pavlíček, "Deployment of A-globe multi-agent platform," in *Proc. AAMAS*, 2006, pp. 1447–1448.

[37] M. Eichberg, "BAT2XML: XML-based Java bytecode representation," in *Proc. 1st Workshop Bytecode Semantics, Verification Anal. Transformation*, vol. 141, no. 1. Dec. 2005, pp. 93–107.

[38] S. Mancoridis, B. S. Mitchell, Y. Chen, and E. R. Gansner, "Bunch: A clustering tool for the recovery and maintenance of software system structures," in *Proc. IEEE Int. Conf. Softw. Maintenance*, Aug. 1999, pp. 50–59.

[39] I. Mayk and W. Regli, "Agent systems reference architecture," *Applied Communications and Information Networking Program*, U.S. Army CERDEC and Drexel University, Camden, NJ, USA, Tech. Rep. V1.0, Jun. 2011.

[40] J. Archibald, J. Hill, F. Johnson, and W. Stirling, "Satisficing negotiations," *IEEE Trans. Syst., Man, Cybern. C, Appl. Rev.*, vol. 36, no. 1, pp. 4–18, Jan. 2006.

[41] J. Archibald, J. Hill, N. Jepsen, W. Stirling, and R. Frost, "A satisficing approach to aircraft conflict resolution," *IEEE Trans. Syst., Man, Cybern. C, Appl. Rev.*, vol. 38, no. 4, pp. 510–521, Jul. 2008.

[42] D. Sislak, P. Volf, M. Pechoucek, and N. Suri, "Automated conflict resolution utilizing probability collectives optimizer," *IEEE Trans. Syst., Man, Cybern. C, Appl. Rev.*, vol. 41, no. 3, pp. 365–375, Jan. 2011.

[43] P. Beaumont and B. Chaib-draa, "Multiagent coordination techniques for complex environments: The case of a fleet of combat ships," *IEEE Trans. Syst., Man, Cybern. C, Appl. Rev.*, vol. 37, no. 3, pp. 373–385, May 2007.

[44] R. Kakehi and M. Tokoro, "A negotiation protocol for conflict resolution in multi-agent environments," in *Proc. Int. Conf. Intell. Cooperative Inform. Syst.*, May 1993, pp. 185–196.

[45] T. Sugawara, S. Kurihara, T. Hirotsu, K. Fukuda, and T. Takada, "Predicting possible conflicts in hierarchical planning for multi-agent systems," in *Proc. 4th Int. Joint Conf. Autonomous Agents Multiagent Syst.*, 2005, pp. 813–820.

[46] F. Lopes, N. Mamede, A. Q. Novais, and H. Coelho, “Towards a generic negotiation model for intentional agents,” in *Proc. 11th Int. Workshop Database Expert Syst. Appl.*, 2000, pp. 433–439.

[47] E. Oliveira, F. Mouta, and A. Rocha, “Negotiation and conflict resolution within a community of cooperative agents,” in *Proc. Int. Symp. Autonomous Decentralized Syst.*, 1993, pp. 421–427.

[48] I. Mayk, W. Regli, D. Nguyen, M. Mai, A. Chan, T. Urness, B. Goren, S. Kizenko, S. Randles, Z. Jastrebski, J. Ruschmeyer, Lex Lehman, M. McCurdy, D. Millar, I. Simmons, C. Cannon, J. Kopena, A. Patwardhan, G. Tassone, J. Lindquist, G. Jewell, R. Forkenbrock, M. Nicholson, F. Koss, R. Wray, J. Bradshaw, and J. Lott, “Tactical information technology for assured network operations (TITAN) information dissemination and management (ID&M) for battle command support services,” presented at *27th U.S. Army Sci. Conf.* Orlando, FL, USA, Dec. 2010.

[49] I. Mayk, W. Regli, D. Nguyen, M. McCurdy, I. Simmons, M. Mai, A. Chan, T. Urness, B. Goren, S. Kizenko, S. Randles, Z. Jastrebski, J. Ruschmeyer, L. Lehman, D. Miller, C. Cannon, J. Kopena, A. Patwardhan, G. Tassone, J. Lindquist, G. Jewell, R. Forkenbrock, M. Nicholson, F. Koss, R. Wray, J. Bradshaw, and J. Lott. “Net-centric information knowledge management and dissemination for data-to-decision C2 applications using intelligent agents and service-oriented architectures,” in *Proc. Military Commun. Conf.*, Nov. 2011, pp. 1568–1573.

William C. Regli (A’03–M’03–SM’06) received the B.S. degree in computer science and mathematics from Saint Joseph’s University, Philadelphia, PA, USA, in 1989, and the Ph.D. degree in computer science from the University of Maryland, College Park, MD, USA, in 1995.

He is currently a Professor of computer science at Drexel University, Philadelphia, PA, USA, with joint appointments at the Department of Mechanical Engineering, Electrical and Computer Engineering, and Department of BioMedical Engineering and Health Systems. His research has been sponsored by a wide variety of organizations. He holds five patents and has authored or co-authored more than 250 technical publications. His current research interests include several computer science and engineering fields such as artificial intelligence, solid modeling and graphics, computer-aided design/computer-aided manufacturing integration, mechanical design, and wireless networks.

Dr. Regli was a recipient of many awards, including the National Science Foundation CAREER Award, the National Research Council Post-Doctoral Award, the National Institute of Standards and Technology Special Service Award, and the Drexel College of Engineering Research Award. He was a co-recipient of the Army’s 2006 International Collaboration and the Institute for Defense and Government Advancement’s Best Network-Centric Warfare Program Award. He is a Life Member of the Association for the Advancement of Artificial Intelligence and Sigma Xi. He is a Senior Member of the Association for Computing Machinery.

Israel Mayk (S’80–M’80–SM’88) received the B.A. degree in physics from Rutgers University, Newark, NJ, USA, in 1970, the M.Sc. degree in nuclear physics from the Weizmann Institute of Science, Rehovot, Israel, in 1973, and the Eng.Sc.D. degree in electrical engineering from the New Jersey Institute of Technology, Newark, NJ, USA, in 1985.

He is currently an Electronics Engineer/Research Scientist and a Technical Manager with the Command, Power and Integration Directorate, U.S. Army Research, Development and Engineering Command, Communications-Electronics Research, Development and Engineering Center, Aberdeen Proving Ground, Aberdeen, MD, USA. He is responsible for research and development of battle command-knowledge-based decision support systems demonstrations and prototypes, as well as for technology integration and architectures.

Dr. Mayk is a member of the Armed Forces Communication and Electronics Association, the Association of the United States Army, and the U.S. Naval Institute.

Christopher T. Cannon received the B.S. and M.S. degrees in computer science from Drexel University, Philadelphia, PA, USA, in 2010.

He is currently a full-time Research Engineer at the Applied Informatics Group, Drexel University. His current research interests include multiagent systems, distributed systems, test and evaluation, and knowledge representation.

Joseph B. Kopena was previously a Graduate in computer science at Drexel University, Philadelphia, PA, USA.

He is currently the Founder of Bellerophon Mobile, a company focusing on intelligent decision making in tactical wireless environments. His current research interests include knowledge representation and wireless networking.

Robert N. Lass (S’06) received the Undergraduate degree from Drexel University, Philadelphia, PA, USA, in 2003, where he is currently pursuing the Ph.D. degree.

He is a Graduate Research Fellow at the Applied Informatics Institute, Drexel University. His current research interests include constraint reasoning, distributed systems, and mobile ad hoc networks.

William M. Mongan (S’06) received the Undergraduate degree in computer science from Drexel University, Philadelphia, PA, USA, in 2005, the M.Sc. degree in science of instruction from the School of Education, Drexel University, in 2008, and the M.Sc. degree in computer science from Drexel University, in 2008.

He is currently with the Department of Computer Science, Drexel University. He was engaged in computer science education and engineering education. He has taught and volunteered with students in grades five through 12. He is an Instrument-Rated Private Pilot of single- and multiengine airplanes. His current research interests include software architecture and composition, service-oriented architectures, agent-based systems, and program comprehension through software engineering.

Mr. Mongan was a fellow of the National Science Foundation GK-12 for two years. He holds a Secondary Mathematics Teaching Certification in Pennsylvania, and has been a member of the School Board Technology and Grant-Writing Committees.

Duc N. Nguygen received the B.S. degree in mathematics from Carnegie Mellon University, Pittsburgh, PA, USA, and the M.S. degree in computer science from Drexel University, Philadelphia, PA, USA.

He is currently a Research Engineer with the Applied Informatics Group, College of Information Science and Technology, Drexel University. His current research interests include artificial intelligence, computer networks, and software engineering.

Jeff K. Salvage received the B.S. and M.S. degrees in computer science from Drexel University, Philadelphia, PA, USA.

He is currently a Senior Lecturer at the Department of Computer Science, Drexel University, Philadelphia, PA, USA. His expertise is in database systems and software design. He is the author or co-author of many books on a variety of subject matter within and outside of computer science. He has been involved in both academia and the corporate world.

Evan A. Sultanik (S'05) received the B.S. and M.S. degrees in computer science and the B.S. degree in mathematics from Drexel University, Philadelphia, PA, USA. He is currently pursuing the Ph.D. degree in the Drexel Applied Communications and Information Networking Program, Drexel University, Philadelphia, PA, USA.

His current research interests include distributed artificial intelligence, ad hoc networking, metrics, approximation algorithms, mobile and multiagent systems, simulation, and constraint reasoning.

Dr. Sultanik was a recipient of a number of fellowships and awards, including the Hill and Koerner Fellowships.

Kyle Usbeck received the B.Sc. and M.Sc. degrees in computer science from the College of Engineering, Drexel University, Philadelphia, PA, USA. His thesis, Network-Centric Automated Planning and Execution, investigates a novel method of generating, executing, and monitoring automated plans in dynamic, heterogeneous network environments.

He is with Raytheon BBN Technologies as a Software Engineer at the Distributed Systems Group. There, he has contributed to projects involving quality of service over airborne networks, spatial computing, and electro-mechanical system design modification. His current research interests include artificial intelligence, automated planning, multiagent systems, networking, HCI, and mobile computing.