# Development and Specification of a Reference Model for Agent-Based Systems

William C. Regli, *Senior Member, IEEE*, Israel Mayk, *Senior Member, IEEE*, Christopher J. Dugan, Joseph B. Kopena, Robert N. Lass, *Student Member, IEEE*, Pragnesh Jay Modi, William M. Mongan, *Student Member, IEEE*, Jeff K. Salvage, and Evan A. Sultanik, *Student Member, IEEE*

***Abstract*—Agent-based systems have been the object of intense research over the past decade. While great theoretical progress has been made, the software frameworks for creating agent-based systems offer considerable variability in their capabilities and functionality. This paper introduces a reference model for agent-based systems. The purpose of a reference model is to provide a common conceptual basis for comparing systems and driving the development of software architectures and other standards. The Foundation for Intelligent Physical Agents and other groups have advanced a number of agent standards, yet, to date, no comprehensive reference model has been presented for software systems composed of agents. This paper provides an overview of a reference model for agent-based systems. The agent systems reference model is the result of a multiyear effort studying software systems built with agents and software frameworks for implementing these systems. As part of this study, the team applied software reverse engineering techniques to perform static and dynamic analysis of operational agent-based systems. This analysis enabled identification of key common concepts across over one dozen different agent frameworks. To demonstrate its applicability, the reference model is then used to analyze a number of complete agent-based software systems. It is the belief of the authors that the reference model will be an essential prerequisite for future transition, deployment, and integration of agent-based systems.**

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

## I. Introduction

THIS paper provides an introduction and overview to a *reference model* for those who develop and deploy systems based on *agent* technology. The agent systems reference model (ASRM) [1] aims to allow for existing and future agent frameworks to be compared and contrasted as well as provide a basis for identifying areas that require standardization within the agents community. This reference model makes no prescriptive recommendations about how to best implement an agent system, nor the objective to advocate for any particular agent system, framework, architecture, or approach. It:

1) establishes a taxonomy of terms, concepts, and definitions needed to compare agent systems;
2) identifies functional elements that are common in agent systems;
3) captures data flow and dependencies among the functional elements in agent systems;
4) specifies assumptions and requirements regarding the dependencies among these elements.

The ultimate goal of the *ASRM* is to facilitate adoption, adaptation, and integration of agent technologies into systems for use by government and private industry, with a particular focus on applications in military command and control. Reference models achieve this payoff by providing appropriate abstractions, simplifying problem solving, and providing patterns of the solution for software developers [2]. It is essential and commonplace to create reference models in all fields of knowledge.

*Statement of need:* The need for a reference model for agent systems is historically similar to the need for communications standards. In the early 1980s, communications systems were proprietary in nature; consequently, there was a divide between communications devices and computer systems. The proposed solution was to establish an open system architecture: an $n$-layered approach to standardize communications systems [3]. Eventually, a seven-layer model was established that has withstood the test of time. The growth of these communications systems prior to the development of the reference model is comparable to the present level of interest in *agent frameworks*. In the context of this paper and the reference model, *agent frameworks* are the libraries and application programming interfaces (APIs) for the construction of agents and agent-based systems. Enough frameworks exist right now and are openly available for inspection to lay the groundwork for such a reference model. In the course of this paper, the investigators studied over one dozen fielded agent frameworks, including those in [4]–[8], with A-Globe [9], enhanced mobile agent architecture (EMAA) [10], Java agent development environment (JADE) [11], NOMADS [12], and the Control of Agent Based System (CoABS) Grid [13] having presentation herein.

It is our belief that the reference model is compatible with all existing agent frameworks and will be applicable to all those in the future. The ASRM is designed by extracting commonalities between existing frameworks and applying existing agent research. The ASRM provides a common ontology for those

Manuscript received July 8, 2008; revised November 17, 2008. First published May 15, 2009; Current version published August 19, 2009. This paper was recommended by Associate Editor R. Brennan.

W. C. Regli, C. J. Dugan, J. B. Kopena, R. N. Lass, W. M. Mongan, J. K. Salvage, and E. A. Sultanik are with the Department of Computer Science, Drexel University, Philadelphia, PA 19147 USA (e-mail: regli@drexel.edu; cjd48@cs.drexel.edu; tjkopena@cs.drexel.edu; urlass@cs.drexel.edu; wmm24@drexel.edu; jks29@drexel.edu; eas28@cs.drexel.edu).

I. Mayk is with the U.S. Army Research, Development and Engineering Command (RDECOM), Communications-Electronics Research, Development and Engineering Center (CERDEC)/C2D, Fort Monmouth, NJ 07703 USA (e-mail: israel.mayk@us.army.mil).

P. J. Modi, deceased, was with the Department of Computer Science, Drexel University, Philadelphia, PA 19147 USA.

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.2009.2020507

frameworks studied and those yet to be built. Previous systems benefit from this model since the advantages and disadvantages of each system become clear. Developers of future frameworks will have a blueprint to follow, forming the basis on which all agent systems—past, present, and future—are compared.

With the ASRM, it becomes possible to specify applications without the need to consider which framework is being used. In fact, the applications should be supported across multiple frameworks that adhere to these guidelines. System engineering tasks also benefit from the reference model because the information used to construct the model was gathered using both common and specially developed analysis techniques. Previously built systems were analyzed and compared in order to piece together a common model. These techniques are applicable to the analysis of almost any system since they, too, are built piece by piece. The specially developed techniques can also be applied in a similar fashion. Further, the techniques themselves can be studied and improved. Agent frameworks are organizationally similar to operating systems, so these reverse engineering techniques also apply in this realm.

*Contributions of this paper:* There are two key contributions of this paper. First, and most directly, is that the paper introduces terminology and concepts to form a reference model for agent-based systems. Note that, due to page limitations, this paper can provide only an introduction and overview to the ASRM. For more detail—including over 200 pages of terminology use cases and examples—readers are referred to the complete ASRM [1].

A second, more indirect, result of this research is our novel approach to the construction of this reference model using software reverse engineering techniques. By performing reverse engineering techniques on existing (open-source and proprietary) agent systems and frameworks, one obtains the software modules that comprise the subject systems. Underlying assumptions on the part of agent framework developers about "what is an agent" and how agents should interact become transparent. Data are produced allowing the documentation and understanding of legacy software systems and for verification of existing software documentation. These data are further abstracted to obtain this abstract "essence" of the systems. By grouping, abstracting, and querying these data in different ways, quantitative, objective information about the design of agents and agent-based systems is obtained.

This is useful in several ways. First, reverse engineering techniques allow not only for identification of components within the reference model, but also to identify both structural and behavioral similarities to the reference model. Agent systems can be automatically observed at runtime and analyzed to find exactly which components correlate with particular features offered by the agent framework. The result is a set of components that are mappable directly to the reference model, thus validating its relevance to existing agent systems.

*Organization of this paper:* This paper is organized as follows. First, an introduction to the background of reference modeling and a review of agent systems literature are provided. This is followed by a description of the technical approach employed to produce the reference model, including a list of definitions. The layers of the reference model itself are then introduced, followed by descriptions of the individual functional concepts therein. Finally, the utility of the model is exampled through mappings of existing agent frameworks to the reference model and also domain-specific case studies.

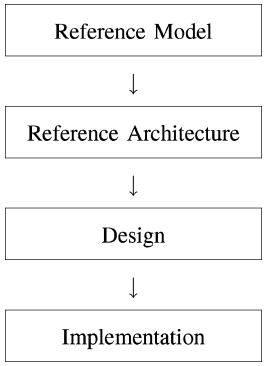


Fig. 1. Role of a reference model: Drives creation of one or more reference architectures, which drive the creation of one or more designs, which, in turn, drive the development of one or more implementations.

## II. Background

This section provides an introduction to reference modeling and background on multiagent systems (MASs).

### A. On Reference Modeling

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. A reference model can drive the implementation of multiple architectures in the same way as that a reference architecture could drive multiple designs, or a design could drive multiple implementations (see Fig. 1).

The reference model provides a common ontology, innovative and practical system engineering techniques, and software development guidance. All of these ideas support evolving agent frameworks and application program interfaces (APIs)—past, present, and future. If the ASRM is used correctly, the result will be specification of and discourse on independently developed software agents and agencies regardless of their environment.

*1) Examples of Reference Models:* The basis for this effort is to follow the approach advocated by the International Standards Organization (ISO) for the development of reference models. In addition, this effort aims to be compatible with the reference models being developed for the Federal Enterprise Architecture.[1] This paper draws heavily on the reference modeling approaches utilized by several approved international standards, in particular ISO 14721:2003, also known as Consultative Committee for Space Data Systems (CCSDS) 650.0-B-1, the "Reference Model for an Open Archival Information System (OAIS)," and ISO/IEC 7498-1:1994, the "Open Systems Interconnection Basic Reference Model."

[1]http://www.feapmo.gov/

An excellent example of an existing reference model is the ISO Open Systems Interconnection (OSI) reference model, which describes a seven-layered network framework for implementing protocols. OSI only describes the abstract functional layers of the network, and does not impose standards or protocols that are to be used at each layer. One could choose to use Transmission Control Protocol (TCP)/IP, Appletalk, or even create their own protocol for each layer, while still remaining true to the OSI model.

*2) Related Reference Models and Standards for Agents:* In the area of agent systems, a reference model for mobile agent systems was constructed in [14]. While superficially similar to the ASRM (most of the definitions for terms, relationships, and abstract entities are compatible with the ASRM), its main focus is on comparing and evaluating different *mobile* agent systems. In addition, the model is more prescriptive of software architecture than the ASRM, which is independent of the particular software architecture for agents. For example, a set of minimum feature requirements (e.g., one of the required components, the *agent execution system*, supports mobility, communications, agent serialization, and security) is presented in [14].

Standards such as those of the Foundation for Intelligent Physical Agents (FIPA), Knowledge Interchange Format (KIF), Knowledge Query Manipulation Language (KQML), and even some nonagent specific standards have a place here, some of which may be used in conjunction with the reference model. The reference model defines the required *existence* of components; standards prescribe how they are *designed*.

Some may see a resemblance between the FIPA Abstract Architecture [15] and the ASRM; however, the reference model is a further abstraction of an abstract architecture. The ASRM defines terms, describes concepts, and identifies functional elements in agent systems. The goal is to allow people developing and implementing agent systems to have a frame of reference to discuss agent systems. The FIPA Abstract Architecture describes an abstract architecture, with the intent of enforcing interoperability between conforming agent systems.

### B. Review of Agent Systems Literature

In addition to the software analysis results, this reference model is a reflection of a wide variety of research publications on agent technology and systems. Information was gathered from many sources including journals, magazine articles, conference proceedings, textbooks, white papers, and manuals for agent frameworks. Due to the breadth of subjects classified under "agent research," a comprehensive background study is beyond the scope of this paper and would be duplicative of excellent survey results previously published. This section provides a brief "survey of surveys" to provide a context for the ARSM. It is organized by the subject where the actual categories are purposely broad and the scope of each is the entire agent world. This section does *not* attempt to make a universal observation about the agent field or any particular subset. These surveys can be categorized as follows.

1) *Analysis of the agent programming paradigm:* Surveys of this type comment on the usefulness of agent frameworks or architectures. Applications are sometimes provided as evidence, and comparisons are usually made with other programming paradigms.
2) *Agents as methodology:* These surveys explain how to conduct agent research and how the results can be applied to the real world.
3) *Status of agent research:* Surveys that fall into one of three subcategories can be found here: historical overview, present trends, or predictions for the future.
4) *Agent textbooks:* Textbooks attempting to distill agent research into a form suitable for undergraduate or graduate education have appeared in recent years. They may serve a variety of purposes, so they are included in their own category.

Several key references in each of these areas are introduced later. While there are many other possible articles that could be cited in each category, those presented later provide a reasonable starting point for anyone interested in understanding the agent field as a whole.

*1) Agents as Software Paradigm:* Chess *et al.* [16] describe the lifecycle of an agent and how it migrates internally through the API and externally through the network between nodes. Etzioni and Weld [17] attempt to formally define an agent by listing some desirable characteristics: it should be autonomous, exhibit temporal continuity, have its own character, be able to communicate, be adaptive, and be able to migrate. Many information agents exhibit these characteristics, and Etzioni analyzes SoftBot as an example. Ghezzi and Vigna [18] show benefits of using agent technologies by comparing and contrasting separate implementations. In particular, they highlight the differences by comparing some implementations with the client–server model and a remove evaluation design. Agents can include mobile code and there are three categories used to compare the implementations of mobile agents: message-based transmissions, strong mobility, and weak mobility. Fuggetta *et al.* [19] explain mobile code and present a framework for understanding such mobile code and give a wide range of example implementations. Kotz *et al.* [20] analyze the many barriers that agent research must overcome before it can be differentiated from mobile code.

In a seminal work, Huhns and Singh [21] edited a collection of documents highlighting key characteristics of agent thinking, emerging applications, architectures and infrastructures, and theoretical models. These documents are by various authors and from various sources, but represent the "best synthesis of current thinking." Apart from the documents, the authors give a good overview of agent-related terms: agents, systems, frameworks, environments, and autonomity. The authors also provide an operational definition of computational agency in [22].

Gray *et al.* [23] establish the major advantages for using agents. They admit that for every program implemented using agents, there may be a better alternative; however, agents provide enough generalization to be decent solutions to a wide range of problems. Using agents renders the following advantages: conserved bandwidth, reduction of completion time when executing across a network, reduction in latency across a network, provide more efficient and disconnected operations, providing "automatic" load balancing, and allowing for dynamic deployment of code.

Wooldridge [24] explores the forum of rational agents: those that are capable of performing independent and autonomous actions using "good" decision making. He introduces the belief–desire–intention (BDI) model of rational decision making. Wooldridge then turns his attention to Logic of Rational Agents (LORA). This framework is then used to highlight teamwork, communication, and cooperative problem solving with regard to rational agents.

*2) Agents as Methodology:* Hanks *et al.* [25] describe common problems stemming from the way research is conducted using artificial intelligence (AI) planning-oriented implementations of agent test beds. In particular, Hanks *et al.* promote the idea that many researchers need to realize the extent to which they are working in a controlled environment; experiments merely produce comparisons between environment parameters and how the agent reacts. Likewise, benchmarks and test beds are empirical tools, and thus should be used with caution in the theoretical realm. Fonseca *et al.* [26] recognize the existence of hundreds of agent system implementations that are all quite similar in operation. As the interest in agent research evolves, researchers are restricting their focus to developing their own implementations from the ground up. Fonseca *et al.* believe this approach is awry and that a better solution is to improve upon the existing agent systems. In particular, JADE and ZEUS are examined with their similarities extracted and combined in order to produce a second-generation MAS.

*3) Status Reports on Agents Research:* Nwana [27] explores the definition of an agent. He presents agents as consisting of cooperation, learning, and autonomy. Within each of these possible classifications, there are subclassifications defined such as collaborative agents, interface agents, and so on. Each of these categorizations is described with a hypothesis, goal, motivation, benefits and role, criticism of work in that field, and challenges facing implementation. Hagen *et al.* [28] discuss the impact of agents on various mobile object middlewares. Sycara [29] develops a definition of MAS that involve limited viewpoints, no global control, decentralized data, and asynchronous computation. She also puts forth several problems that are currently faced and will be faced by the developers of such agent frameworks. At a lower level, agents in MAS must contain the ability to reason, organize themselves, and share the workload in order to operate efficiently.

In a seminal work, Jennings *et al.*'s [30] paper attempts to provide "order and coherence" to the field of agent technologies. They begin with a brief overview of agents, highlighting their definition of an agent: "an entity that acquires information, reasons, and reacts." A history of agents is detailed followed by a status report on various fields within agent research, including human–computer interaction, distributed AI, and constraint-based problem solving. An overview of agent-based systems is presented that leads into a discussion about previous implementations, future implementations, and the applications of these systems. The applications are examined at length and include air traffic control, auctioning, video games, medical technologies, and more. Jennings *et al.* conclude by noting that agent-based research is a new field with great potential that will find a variety of applications upon its maturity.

*4) Agents Textbooks:* There are several excellent textbooks in the area of MASs. These include those by Vigna [31], D'Inverno [32], Weiss [33], Wooldridge [34], and, most generally, Russell and Norvig [35]. Milojicic *et al.*'s book [36] is a survey on mobility with a domain that extends to not only agents, but also to processes and computers. The book of Wooldridge [34] is an introductory text to agents and MASs. A brief history is followed by an exposition concerning individual agents. The usual, broad definition of agents is given, and then, reasoning and reacting are examined. The remainder of the book explores collections of agents. In these environments, agents interact in order to solve problems. They can each work independently by solving parts of a problem or work together as a unified group. Specific examples are cited in order to strengthen this notion.

## III. Technical Approach to Agent Systems Reference Modeling

The technical approach taken to create this reference model uses forensic software analysis of existing agent-based systems. An agent-based system may consist of many different kinds of agents operating across a heterogeneous set of computing platforms. Rather than trying to develop a consensus about "what is an agent,"[2] this paper offers a different approach from the largely inconclusive debates of the past: the reference model developed in this paper is based on *static* and *dynamic software analysis* of fielded agent systems. Hence, an agent-based system describes a software platform for *both* building agents *and* supporting their communications and collaboration within systems.

There are many products in the marketplace today that are marketed as agent frameworks from various sources: companies, academia, and the open source community. These agent frameworks have emerged from several large governmental and private research and development programs and were used in the creation of many successful military and commercial systems.

This paper takes a quantitative and evidentiary approach; if it can be built with one of these systems, an artifact might be called an "agent." Anyone building a new agent framework must recreate or reproduce some portion of the components in these frameworks (i.e., to enable communication, agent startup, shutdown, etc.). Hence, by analyzing existing agent frameworks, this reference model documents the existing state-of-the-art for what the community believes is an "agent" by looking at implemented examples.

The result of this analysis is a detailed view of the superset of the features, functions, and data elements in the set of existing agent frameworks. Given that each framework may have slightly different functional components, the reference model describes, at an abstract level, a set of functional components that an agent framework *may* have. It is, however, important to note that the model is not confined to being a description of existing capabilities and platforms—it serves as a basis for situating a set of functional and data elements that anyone may

[2]Or worse, what is an "intelligent" agent.

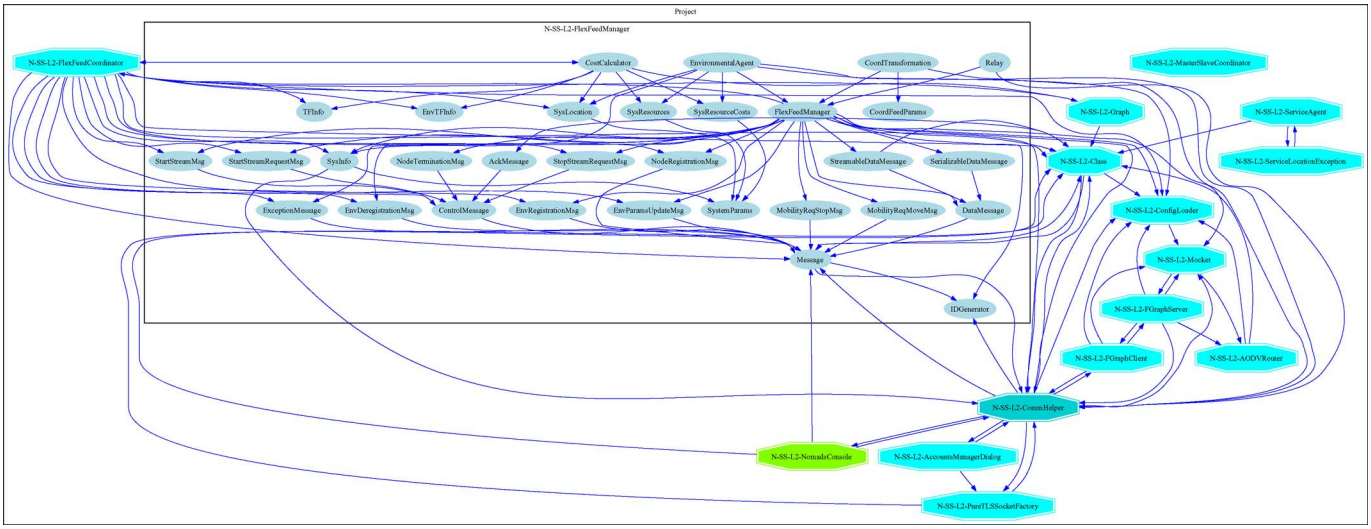


Fig. 2. Example of software analysis on an existing agent framework. In this case, the figure shows a functional breakdown of the FlexFeed subsystem of the NOMADS agent framework. Ovals represent classes, rectangles/octagons represent clusters of classes, and edges represent method invocations.

want or need to have in an agent platform. For example, security for mobile agent code is currently a vastly challenging problem that lacks a satisfactory solution; however, the lack of any established, uniform, and generally accepted security system for mobile agents does not preclude the reference model from including a description of the security functions and facilities that an agent platform should provide. Reference models do not prescribe how functions and systems should be implemented; the ASRM is no different. Given the vast array of tasks envisioned for agent systems, it is not the role of a reference model to account for each possible application architecture.

### A. Creating the ASRM via Reverse Engineering

The typical method for creating a reference model consists of three large phases:

1) capturing the essence of the abstracted system via concepts and components;
2) identifying software modules and grouping them into the concepts and components;
3) identifying or creating an implementation-specific design of the abstracted system.

The approach taken in the ASRM is, however, very different. *Reverse engineering* and *software analysis* methods were employed on existing, fielded agent frameworks and systems to create the ASRM. Reverse engineering is the analysis of software systems by extracting artifacts and functionality from an existing system [37]. Using reverse engineering techniques, one extracts software components and their relationships through automated analysis of a system's source code or runtime behavior. Software components are basic software entities such as classes, collections of classes, and packages. Relationships between components are one or more interactions that exist between software components. By applying these methods, the usual process of creating a reference model is reversed.

In this paper, reverse engineering techniques determine both the structural and behavioral makeup of existing agent systems. The *static analysis* of the software system yields the structural components that exist in the system, and the *dynamic analysis* shows how and when these components are instantiated and used. Moreover, dynamic analysis (sometimes called *behavioral analysis*) shows the runtime interactions between the components found during static analysis [38].

*1) Static Analysis:* Static analysis is the analysis of software using its source code as the primary artifact. The system need not be executing in order to obtain the appropriate data. Instead, source code or intermediate code is inspected to find the software modules, data structures, data flow, and methods and metrics appropriate to the system.

This type of analysis yields many benefits, such as code rewriting, vulnerability detection, finite-state machine verification, and source code repository abstraction. For purposes of the reference model, the primary goal is to use static analysis to produce a data repository from code that can be queried to find the primary software subsystems. This facilitates the transition from analyzing subject systems to identifying software modules that might fit the overall abstract system defined by the reference architecture.

An example of static software analysis applied to a portion of the NOMADS [39] agent framework is shown in Fig. 2. Fig. 2 illustrates a high-level clustering of the class relationships in NOMADS. Class relationships include class inheritance and method invocations between classes. These relationships are clustered by weight to obtain likely subsystems within the architecture. These subsystems are depicted by light-blue octagons. Each cluster expands into a box of class nodes, which are depicted by dark-blue ovals. The expanded cluster is named `N−SS−L2−FlexFeedManager`, because it is a subsystem in which the `FlexFeedManager` class is the "heaviest" node (contains the most relationships) in the cluster. It happens that the FlexFeedManager is an agent management service provided by the framework, and provides interfaces to other framework services [40]. As a result, one might expect this manager to contain relationships with the

message objects and the communications subsystem. This is verifiable in Fig. 2 by observing the relationships between `FlexFeedManager` and the `DataMessage` objects immediately below it. The `FlexFeedManager` also contains relationships to classes within the `N−SS−L2−CommHelper` subsystem, which can be further investigated using dynamic analysis techniques or more static decompositions such as call graphs. In this way, static analysis data often provide insight into which features should be exercised during dynamic analysis or via a specific call trace.

*2) Dynamic Analysis:* Dynamic analysis also collects data on software systems, but it does so by inspecting that system during execution. This analysis varies widely by implementation, but one approach is to build a data repository of program behavior. This repository holds information on data flow, object instantiation, the call graph, interprocess communication, network or filesystem I/O activity, and so on. This analysis assists the production of the reference model by providing more sophisticated justification than is provided from static analysis alone. For example, static analysis relies somewhat on the software architecture of the subject system. If the system contains a lot of "dead code" or other obfuscated constructs, the static analysis results can be inaccurate and deficient in describing the true structure of the system. Dynamic analysis inspects the system as it is running and often breaks the system down into "features." These features can be analogous to the relevant subsystems found during static analysis, such as those found and described in Fig. 2. Moreover, dynamic analysis can obtain data on behavior-specific aspects of the system such as threading and I/O, which could not otherwise be found simply using static analysis techniques. Finally, dynamic analysis can assist in cases where source code is not available for static analysis to be performed. Static analysis decompositions often require that one have access to the source code, or perhaps some equivalent byte code. However, it is often the case with legacy systems that only the binaries are available, and thus, it is not possible to extract architectural information. However, debuggers, tracers, and platform-dependent dynamic analysis tools trace the execution of a software system and (if the information is available) provide a call trace. This can help reconstruct the architecture of certain features of a system by observing how the system behaves when that feature is executed. The methods and objects invoked by a feature that is observed during dynamic analysis are called a slice of the system. When static analysis is also feasible for a system, slices can be mapped to the static architecture decomposition to identify which subsystems are exercised during a trace.

An example of dynamic software analysis is applied to a portion of the Jade agent framework to show feature extraction. By mapping packages and the order that the packages are invoked, a temporal view of a scenario demonstrating an invocation point of a functional component is generated, as shown in Fig. 3. In this example, agent mobility in Jade is exercised, thus extracting the associated runtime trace. This reveals dependencies on the Jade Agent Management System (AMS) via a call to the `AMSService` and `getAMS()`. The agent is migrated via a call to `doMove`, and named on the new host via a call to `getName`.

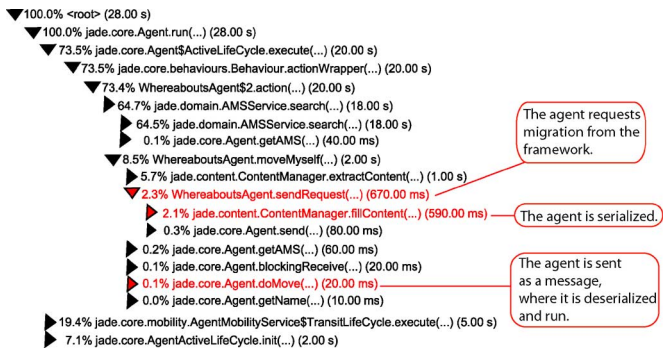


Fig. 3. Jade mobility runtime trace.

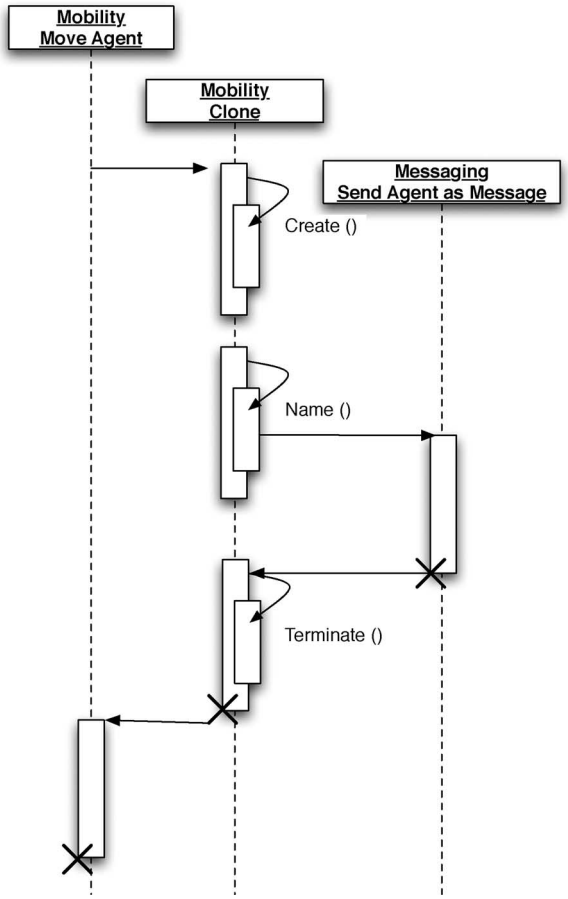


Fig. 4. Jade mobility process view activity diagram.

For the reference model, these reverse engineering data are used as the basis for constructing a high-level view of an agent system. A structural and behavioral overview of the system is constructed first from the concrete agent framework (in this case, Jade), and then, an aggregate structural and behavioral overview is created from this and other agent framework reverse engineering data.

The behavioral view is constructed by mapping packages and the order that the packages are invoked; a temporal view of a scenario demonstrating an invocation point of a functional component is generated, as shown in Fig. 3. The key objects and methods are highlighted based on the code snippets to generate an activity diagram, as shown in Fig. 4.

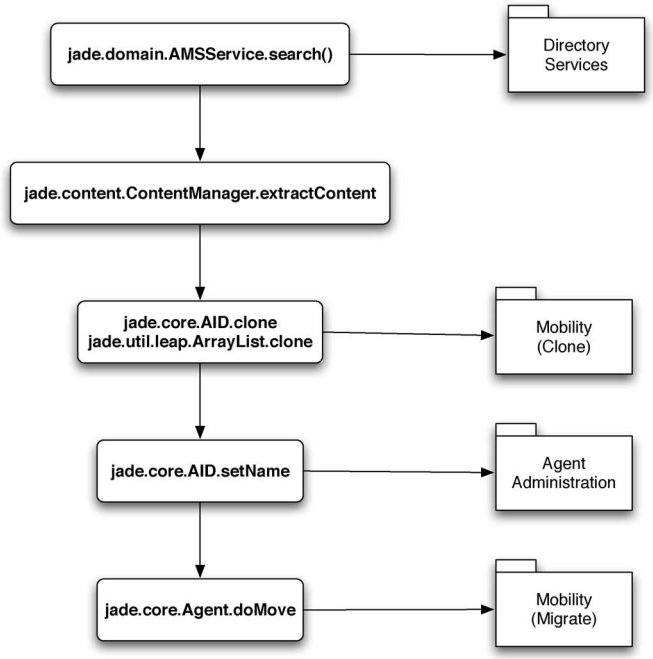


Fig. 5. Jade mobility logical view package diagram.

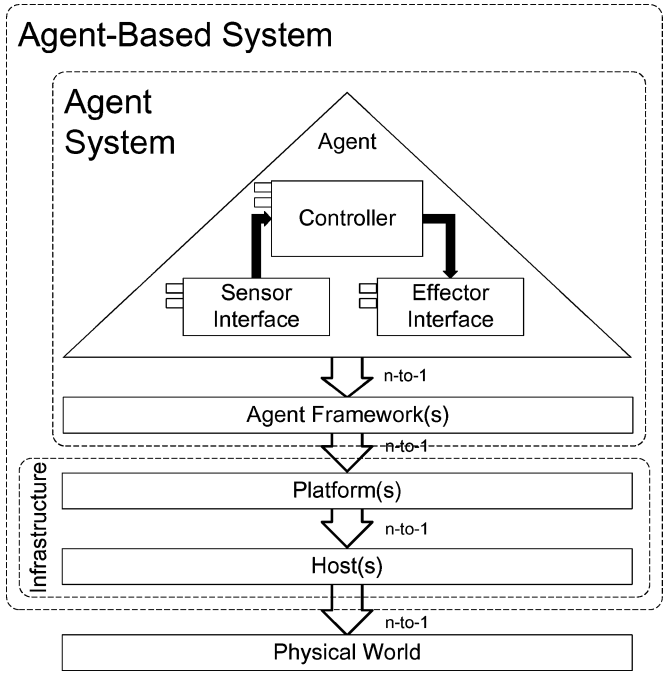


Fig. 6. Abstract model of an agent system. Such systems decompose into several layers of hardware and software that provide an operating context for agents, situated computational process that sense and affect their environment. Note that the relationships across layers may be $n$-to-1.

To construct the structural view, we identify cloning and serialization as the "helper" functions, but most importantly, we discover that Jade's mobility code largely uses the same classes as the messaging functional concept. This indicates that Jade mobility is implemented using the messaging code base. The primary calls that were observed in the runtime trace shown in Fig. 3 inform the activity diagram activities and process in Fig. 4, as well as the package diagrams depicted in the logical view in Fig. 5.

*3) Typical Results of Reverse Engineering Analysis:* Reverse engineering analysis yields components and relationships that provide the grounding for the construction of the reference model, as described in Sections IV and VI. Static analysis techniques are used to obtain structural information about software system features from source code [41]. Moreover, runtime analysis enables design extraction by providing feature traces that specify what source code is exercised during a feature's execution [42].

Two components might interact via a method call, by sharing data, or by aggregating one another through class inheritance or implementation; these are all examples of component relationships. Components and relationships are often depicted using an *entity-relationship (ER) graph*, in which components are referred to as entities or nodes and relationships are referred to as edges between components.

One can further extract these interrelationships by identifying the level of coupling (the amount of relationships) and the type of relationships that exist between components. It is often the case in software systems that components are relatively loosely coupled, but are locally tightly coupled. In other words, most components do not depend directly on one another on the whole, while related components interact to achieve their common functionality.

Collections of relationships, called *clusters*, are formed by grouping components with only a high degree of coupling. This process may be repeated any number of times by further grouping entire clusters based on their coupling. Software analysis tools exist to extract and abstract data from systems in this way. The end result is usually a hierarchical depiction of the software system, in which clusters of clusters of components are shown.

These data may be static components such as classes and call graphs or it may be dynamic components such as instantiation and data flow. In either case, the hierarchical result is ideal for identifying subsystems that exist within a software system, such as disk access and graphic display, as well as layers (collections of subsystems, or clusters of clusters) that comprise the system's architecture. For example, disk access and RAM access might be combined as part of a larger memory management layer, and so on. By appropriately abstracting these layers, one uses reverse engineering techniques to make a good hypothesis to a generic architecture (called a *reference architecture*) that comprises a class of software systems, such as operating systems. In addition, reverse engineering (RE) validates and identifies discrepancies between this reference architecture and existing systems.

## IV. Agent System Reference Model: Initial Definitions

*Software agents*, sometimes called intelligent agents or simply "agents," are *situated* computational processes—instantiated programs that exist within an environment that they sense and affect. Fig. 6 portrays an abstract model of an agent and its relationship with the system and environment in which it exists. An agent actively receives *percepts*, signals from the environment, through a *sensor interface*. Though its response

need not be externally observable at all times, an agent may take actions through an *effector interface* that can manipulate and affect that environment. Importantly, the model does not commit sensor and effector interfaces to specific hardware or software structure and form, but rather generically as data flow in and out of an agent.

Being situated in an environment is a key property of agents, whether that environment be a virtual (i.e., a file system or the World Wide Web) or a real world setting (i.e., a computer network, a robotic system, or an image understanding system). Although the focus of this paper is on software agents, this does not preclude the possibility that an agent or collection of agents may be embodied in the physical world, e.g., a sensor monitoring system or a robot controller. In addition to being situated in an environment, one or more of the following properties hold for any agent.

1) *Autonomous:* Agents may perform their own decision making, and need not necessarily comply with commands and requests from other entities.
2) *Proactive:* Agents need not wait for commands or requests and may initiate actions of their own accord.
3) *Interactive:* Agents may observably respond to external signals from the environment, e.g., reacting to sensed percepts or exchanging messages.

Although many agents possess two or all of these properties, it is possible to construct agents possessing one but not the others. However, software that is not situated or does not hold one of these properties forms a different class of software from agents. In particular, *services*, which are computational processes that exist to provide functionality for use by other processes, do not necessarily exhibit these properties. While by definition of interactive property, services may have no significant ties to an external environment. They are also infrequently associated with autonomy and proactivity. While an agent may be or provide a service, a service is not, in general, an agent. Other properties that may hold of an agent include the following.

1) *Continuous:* Agents are typically a long-lived thread of execution. They are not spawned and terminated for each individual task. As described later, specific agent technology may provide support for preservation and resuscitation across restarts and other events.
2) *Social:* Many agents interact significantly with other agents in achieving their tasks, which is a specialization of interactive agents. Social agents may be further classified with respect to the relationship between their implicit and explicit priorities, preferences, and actions versus those of other agents. Basic divisions along these lines include *self-interested*, *adversarial*, and *cooperative* agents. Such agents may utilize many protocols and forms of discovery, coordination, communication, and negotiation in their interactions, as discussed in Section VI.
3) *Mobile:* Some agents are not static, fixed features of the operating environment. Robots may physically move in the world; software agents may *migrate* between computing devices—temporarily pausing execution, transferring to another host, and continuing execution there. Mobility is further classified and described in Section VI.

Section V-C discusses several other properties in the context of multiple agents and overall system applications. These include the level of reasoning individual agents conduct and the sophistication of the tasks they perform.

## V. Agent System Reference Model: Concepts and Layers

This section begins to formalize and describe concepts for agent systems. It places agents within the context of required infrastructure, and the larger computational and world environment. Several diagrams are used throughout this section and the remainder of the diagram to define and explain these models.

### A. Infrastructure for Building and Supporting Agents

As computational processes, agents do not exist on their own but rather within computing software and hardware that provides them mechanisms to execute. Many agent implementations also require substantial libraries and code modules. Further, agents frequently possess properties not found in traditional software, such as mobility. Development and implementation of such software require significant infrastructure to provide core functionality that agents may use in conducting their tasks.

An *agent-based system* comprises one or more agents designed to achieve a given functionality, along with the software and hardware that supports them. It is comprised of several layers, as shown in Fig. 6, and is described as follows.

1) *Agents* implement the application; they achieve the intended functionality of the system.
2) *Frameworks* provide functionality specific to agent software, acting as an interface or abstraction between the agents and the underlying layers. In some cases, the framework may be trivial or merely conceptual, for example, if it is merely a collection of system calls or is compiled into the agents themselves. At this extreme, the framework is considered "null" or empty, such as in the case where agents are programmed directly into hardware. This is consistent, since implementing their agents in such a way would be a conscious decision of the agent system designers. A virtual machine is an example of an agent framework in the other extreme.
3) *Platforms* provide more generic infrastructure from which frameworks and agents are constructed and executed. Items such as operating systems, compilers, and hardware drivers make up the platforms of an agent system.
4) *Hosts* are the computing devices on which the infrastructure and agents execute, along with the hardware providing access to the world. This may range from common disk drives and displays to more specialized hardware such as GPS receivers or robotic effectors.
5) *Environment* is the world in which the infrastructure and agents exist. This may include physical elements, such as the network connections between hosts, as well as computational elements, such as Web pages the agents may access.

An *agent system* is simply a set of frameworks and agents that execute in them. An *MAS* is an agent-based system that includes

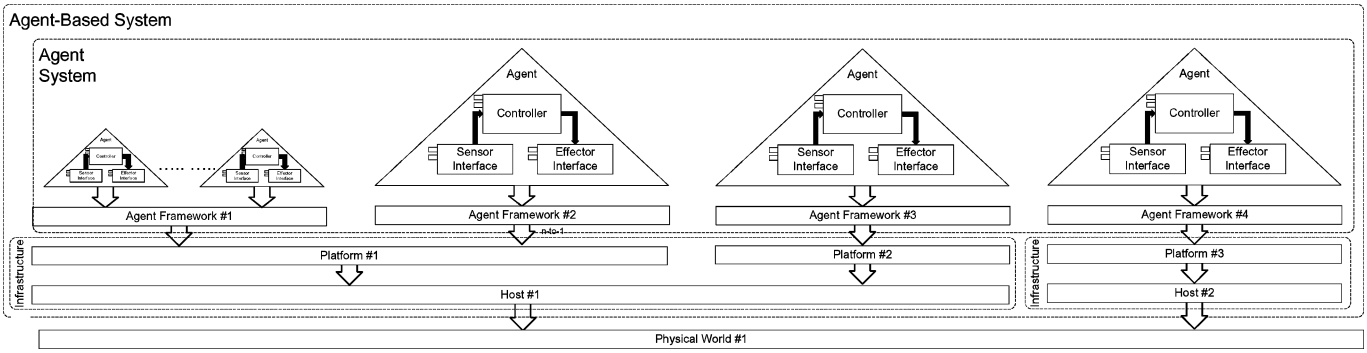


Fig. 7. Agents and agent frameworks are often part of larger systems. Such larger systems are called agent-based systems and encompass all of the different agents, frameworks, platforms (along with their nonagent software), and hosts needed to deliver the functionality of the system.

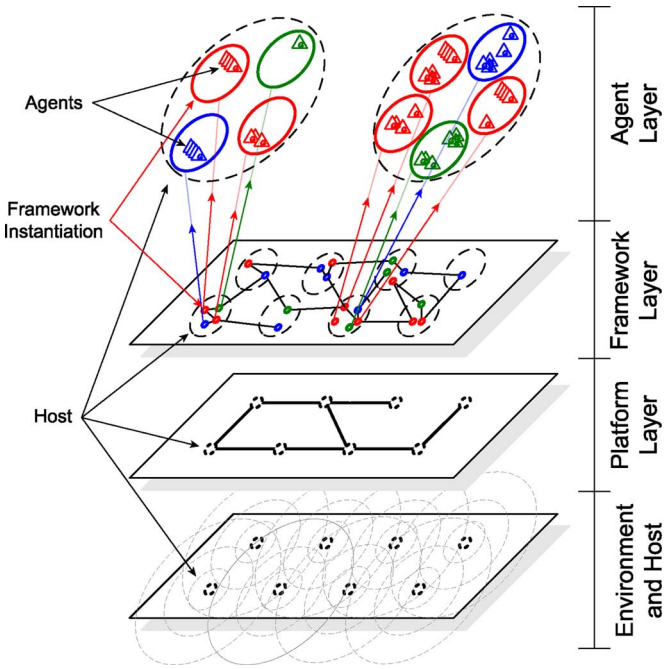


Fig. 8. Agents are depicted as computational processes running within frameworks supported by platforms and executing on hosts operating together on some network.

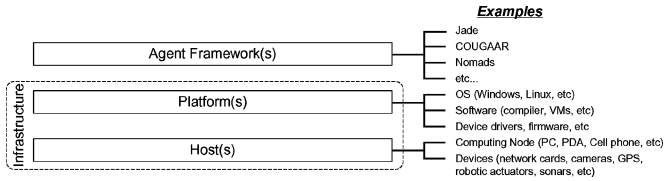


Fig. 9. Examples of technologies (circa 2006) mapped onto different layers of agent systems, as presented in Fig. 6.

more than one agent. Such systems may consist of many agents running within a single framework instantiation, or in different frameworks, on different hosts, etc. A conceptual example of an agent-based system is shown in Fig. 7 that extends over several hosts. Fig. 8 gives another example of devices in the agent system connected at the host layer via wireless networking, and transmitting and receiving signals in the environment of the physical world. With respect to the ASRM, communications are abstracted at the platform layer by the operating system and network software, e.g., routing tables. At the framework layer, each platform has one or more executing frameworks. Each framework instantiation then may be associated with many currently executing agents in the agent layer.

Taken together, the hosts and platforms of an agent system define the *infrastructure* that provides fundamental services and operating context on which frameworks are constructed. Frameworks and infrastructures mediate between agents and the external environment, and therefore between agents, thus providing for both execution and access to the world. Fig. 9 provides several examples of current technology mapped onto these layers.

In general, this paper does not distinguish between instantiations of elements and their type except when it is important. For example, the distinction between an agent program's source code and an executing instantiation of that program is only drawn when necessary. However, each agent system layer explicitly supports multiple entities above it—a framework may be executing a multitude of agents, a platform may contain several frameworks, and so on.

### B. Communication Among Agents

Communication among agents is a critical aspect of many agent systems. As such, the existing OSI (ISO 7498:1984) reference model is applicable to describing communication among and between agents. There are several distinct ways in which this mapping can be made. In the context of this paper, agents are situated software processes communicating within the context of a larger system. This means that agents are situated within a system and make use of its communication components.

Fig. 10(a) shows the established OSI seven-layer communications model. In this model, the following layers are present.

1) The physical layer (layer 1) defines all the electrical and physical specifications for devices and their major functions.
2) The data link layer (layer 2) provides the functional and procedural means to transfer data between network entities and to detect and possibly correct errors that may occur in the physical layer.
3) The network layer (layer 3) provides the functional and procedural means of transferring variable-length data sequences from a source to a destination via one or

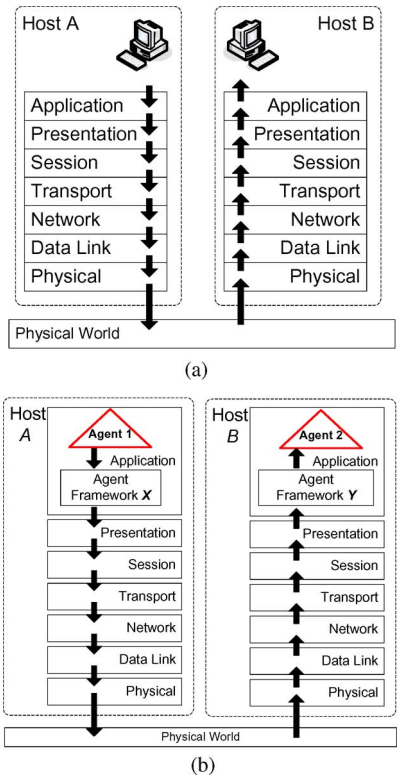


Fig. 10. Agent systems often fit nicely within the application layer of the OSI reference model. A typical example is shown above; however, the agent system is not *required* to reside at the layer depicted. (a) 7 Layer OSI model. (b) Agent systems within the OSI model.

more networks while maintaining the quality of service requested by the transport layer (layer 4). The network layer performs network routing, flow control, segmentation/desegmentation, and error control functions. Traditional hardware routers operate at this layer, for example. The best known example of a layer 3 protocol is the IP.

4) The transport layer (layer 4) provides transparent transfer of data between end users, thus relieving the upper layers from any concern with providing reliable and cost-effective data transfer (e.g., TCP).
5) The session layer (layer 5) provides the mechanism for managing the dialog between end-user application processes.
6) The presentation layer (layer 6) relieves the application layer of concern regarding syntactical differences in data representation within the end-user systems.
7) The application layer (layer 7) provides services that facilitate communication between software applications and lower layer network services so that the network can interpret an application's request and, in turn, the application can interpret data sent from the network.

Fig. 10(b) shows one way (perhaps the most common in practice) that the OSI layered model is related to the ASRM. In this view, agents and agent frameworks exist entirely at the application layer. The agent platform and host (i.e., the agent infrastructure) interfaces with the other layers of the OSI stack, and agents are largely insulated from needing to process information at these layers. It is conceivable that designers of agent systems may wish to have their agents interact with and operate in the OSI layers 1-to-6. This option is not precluded in the current reference model. For designers of such agents (i.e., an agent for OSI layer 2), the agent framework needs to provide APIs or other means for the executing agents to sense and affect operations at these layers.

Alternative mappings between the ASRM and the OSI layers can be made if one considers possible configurations in which agent and agent frameworks assume the responsibilities for the lower layers on the OSI stack. For an extreme example, one could envision each agent as a separate entity that must communicate with other agents, in which case the physical layer of the OSI reference model can be mapped directly to the sensor interface and effector interface of each agent (i.e., Fig. 6), with the functionality of the other layers encoded inside the agent controller. A simple example of this case is in the situation of robotic entities (i.e., the classical vacuum cleaner world) communicating stigmergically through their physical environment.

Other relevant cases for the spectrum of ASRM to OSI mappings include the following.

1) Protocols such as KQML, when implemented at the framework or agent layer, could be used to serve a similar purpose for agents as TCP/IP serves for host communications;
2) Serialization, or other encoding needed for agent mobility, may be considered as a presentation layer functionality and may be provided by the agent framework.
3) Individual agents could encode or encrypt their own messages, hence assuming the functionality of the presentation layer.
4) Agent negotiation and auction protocols are special communications protocols. For certain kinds of auctions, the information flow patterns among the agents could be viewed as message routing. Hence, agents could serve as routers, and these protocols could be mapped to the network layer.

This list of configurations and mappings could be extended into a wide variety of permutations. The conclusion is that there is no one way in which OSI can or should be mapped into ASRM; however, certain mappings, once specified, can help to clarify the context and semantics of agent communications within an agent-based system.

### C. Classifying Agents

The complexity of an agent, and subsequently of an MAS built of many agents, may be viewed along at least two different aspects.

1) *Internal agent complexity:* It is the sophistication of the agent's internal reasoning. Agents may be constructed from simple condition–action rules to elaborate deliberative models.
2) *Operational abstraction:* It is a necessarily informal characterization of the level of problem or task which an agent is intended to address in the world. This may range from simple signal monitoring to human-level problems such as medical or mechanical diagnosis support.

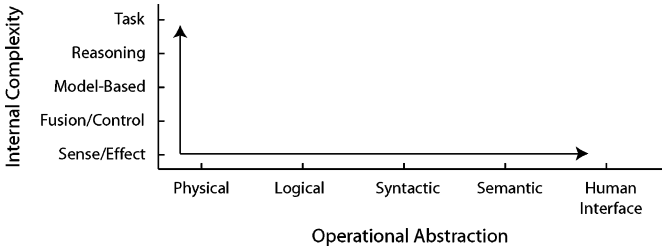


Fig. 11. Matrix of agent internal complexity and level of operational abstraction.

These two aspects outline a classification of agents, as shown in Fig. 11. Every individual agent falls somewhere in this classification. However, the two axes are independent: an agent might perform sophisticated reasoning about a primitive physical task, or might utilize a simple mechanism to address an abstract problem. For example, an agent may perform very sophisticated reasoning about low-level networking decisions, or a very basic rule-based chat agent might answer user queries in "natural language." By classifying agent complexity, these axes can also provide a complexity classification of an MAS through an aggregation of the agents in its community.

An agent of one layer in either axis must possess capabilities of the lower layers. However, agents of disparate layers may interact freely. Further, note that these layers do not imply any particular internal agent structure or implementation preference—an agent's implementation need not be explicitly broken into such layers. Internally, an agent may be built in any fashion and use any reasoning mechanisms. The following two sections describe these axes in more detail.

*1) Internal Agent Complexity:* The internal complexity axis of Fig. 11 is a layering of increasingly sophisticated agent reasoning mechanisms.

1) *Sense/effect* agents are the simplest class of agents, reacting directing to basic environmental stimuli. Examples include agents monitoring smoke detectors or executing stock market limit orders.
2) *Fusion/control* agents integrate multiple environmental stimuli to create a decision. This implies at minimum a method of weighting or prioritizing stimuli in choosing a response. It does not imply a history or fusion of inputs over time. A spam filter agent judging received mail agents by a weighted sum of several taboo word-list scans would be such an agent.
3) *Model-based* agents fuse multiple inputs to produce and evolve a model of the world and its change over time. This may be a simple record of past observations or may incorporate predictions and estimations based on previous actions. For example, a just-in-time inventory monitor may track demand over time to create a predictive model for use in evaluating stock levels and issuing resupply requests.
4) *Reasoning* agents extend model-based mechanisms to plan over multiple actions or perform a sequence of inferences. In addition to memory for tracking and developing world state, reasoning agents implicitly or explicitly possess some notion of goals and a process for determining actions that evolve the current state under the agent's world model to match these goals. Agents involved in multistep service interactions, such as searching and purchasing from online merchants, may have to perform such reasoning in following the protocol.
5) *Task* agents, in addition to having a notion of their own goals, model and reason on the goals of other agents. At the end of this complexity spectrum, a task agent may interact with a set of agents in achieving shared goals. Examples include coordinated robot maneuvers and auction proxy agents.

Of course, more sophisticated reasoning mechanisms are built on more primitive foundations. Agents of a given layer therefore incorporate the underlying layers of internal complexity.

*2) Operational Abstraction:* The operational abstraction axis of Fig. 11 captures the layers of application domains with respect to the external world in which agents may be deployed.

1) *Physical* agents receive raw stimuli from the physical world as their environmental percepts. Examples include agents that monitor physical parameters such as signal strength on wireless networking cards, or agents that receive camera or video input as raw pixel data. Percepts are minimally preprocessed but may be either reacted to in that form, e.g., by a sense/effect agent, or processed and refined, e.g., by a model-based agent. A smoke detector monitor is an example of the former, and a video-processing face detector the latter.
2) *Logical* agents receive primitive, abstracted input from the environment. A user clicking an "OK" button to dismiss a dialog window is an example of such input. An agent that polls Web servers for the existence of or updates to a given Web page and looks for Hyper-Text Transfer Protocol (HTTP) 304 (Not Modified) or 404 (Not Found) responses is an agent of the logical layer.
3) *Syntactic* agents operate on structured or semistructured input with *a priori* fixed meaning and schema. As opposed to a physical agent that receives raw pixel signals from a camera, a syntactic agent may be able to parse a variety of image or video formats. An agent that can read and write Extensible Markup Language (XML)[3] data that may be exchanged with other agents capable of parsing and generating the given schema is another example.
4) *Semantic* agents have an understanding of the primitive elements and composition of data objects they manipulate in their domain. In contrast to syntactic agents, semantic agents may operate on syntactically unstructured data where the elements of the data object's structure or syntax have *a priori* fixed meaning and schema, rather than the data itself. An example is scheduling agents exchanging calendars encoded in the W3C Ontology Web Language (OWL).[4] The syntactic structure of the content of such documents is more free form than under an XML schema, and much meaning may remain implicit for the agent to infer.

[3]http://www.w3.org/TR/REC-xml/
[4]http://www.w3.org/TR/owl-ref/

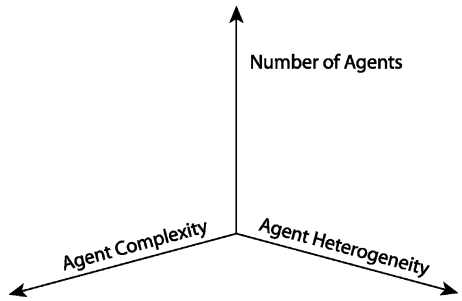


Fig. 12. Dimensions of MAS complexity.

5) *Human interface* or *cognitive* agents are situated in an environment where they work in concert with a human user. Examples include "paper-clip" agents, interactive proof checkers, computer-aided design (CAD)/computer-aided manufacturing (CAM) design aids, and medical diagnosis assistants. Presenting a graphical or other interface to the user is neither sufficient nor necessary for inclusion in the class of human interface agents. While many may include sophisticated cognitive interfaces, some may interact entirely through stigmergy (shared observable effects on the environment). Rather, the key element of a human interface agent is substantial, deliberate user interaction at a significant level in the domain.

The aforementioned classifications are for single individual agents, but have implications for MASs as well. These are further discussed in Section V-D.

### D. MAS Structure

This section provides language and concepts for describing a system comprised of multiple agents. Although systems comprised of a single agent fit within this reference model, many agent systems of interest incorporate several agents where the goals of the agent system are achieved through interactions between the individual agents. When properly designed, these interactions create much more substantial functionality than that of any single agent.

A set of more than one agent will be collectively referred to as a *group*. In addition, a set of groups may also be referred to as a group. Then, the term MAS is used to denote a group of agents plus their supporting frameworks and infrastructure. An MAS may consist of multiple frameworks, executing across multiple hosts and each deploying multiple agents, each of which may have different internal agent architectures of varying complexity.

*1) Dimensions of MAS Complexity:* The primary dimensions for classifying MASs are shown in Fig. 12. These axes position systems based on the number of agent instantiations they include, the internal complexity of these agents, and the number of different types of agents. Classifications of internal agent complexity are discussed in Section V-C. Based on these dimensions, the following terms are defined to describe common types of MASs.

1) *Monolithic system:* It is an agent system consisting of a single agent of high internal complexity. Such systems are close to traditional software, but incorporate notions of agent software such as autonomy, proactivity, and continuity. Many are based on applications of AI topics such as machine learning and logical or probabilistic deduction. Proxy agents that conduct tasks for the user such as scanning the World Wide Web for prices and making purchases to fill given specifications often fall under this category.
2) *Median system:* Many MASs contain a set of moderately complex and heterogeneous agents. This approach to constructing agent systems is common in many domains such as robotics, command and control, and personal assistants. Although it is not required, these systems often employ mechanisms to facilitate coordination, cooperation, and resource sharing that enable efficient and robust goal achievement.
3) *Swarm system:* It is an MAS comprised of many agents, often of a single or several highly similar types, and frequently of low complexity. Individual swarm agents typically act in very simple ways, with interesting overall system behaviors arising as the aggregate of many repeated interactions through the large number of agents present. Swarms provide for robustness and scalability due to a large degree of redundancy and the ability to introduce more agents as necessary with relative ease.

Note that the aforementioned systems are presented as examples and that there are certainly other possible systems architectures for MASs.

*2) Structured Groups of Agents:* The following terms are specializations of the generic group concept, based on the relationship between the goals and behaviors of agents and groups of agents.

1) A *team* is a group with a single or small number of common goals. Frequently, each agent or group plays a particular role in solving a larger problem. These may include leadership and manager roles. However, such structure is not necessary. For example, a team may also be a swarm of homogeneous agents, each contributing in similar ways to the larger functionality.
2) An *organization* is a group that interacts according to some structure, such as a hierarchy. Each agent or group has a goal that may be independent of but not in conflict with the goals of other agents and groups. Frequently, the organization has a common overall goal, with each member working to achieve subgoals of it.
3) A *society* is a group that has a common set of laws, rules, policies, or conventions that constrains behavior. Agents and groups contained therein do not necessarily have any goals in common and may have goals in conflict.
4) An *agency* is a group that specializes in providing expertise or enabling a service in a given domain. There may be constraining policies, e.g., access control mechanisms or resource scaling, and these agents and groups may be competing.

Typically, these terms also have implication on the quantity of agents in the group. For example, teams are often groups within an organization and organization groups within a society.

*3) Communication in MAS Layers:* MASs are comprised of several communication and interaction layers corresponding to

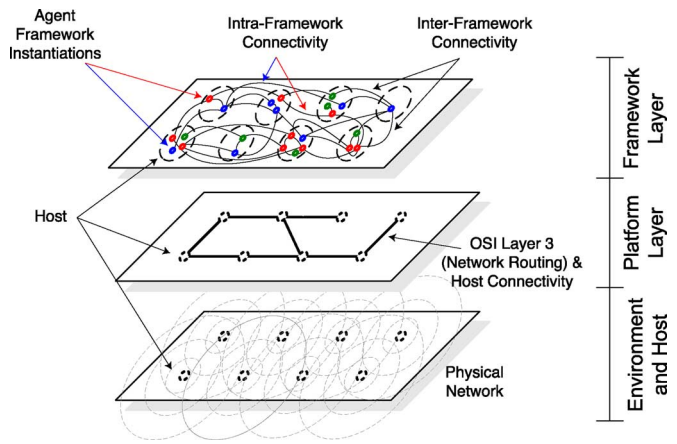


Fig. 13. Communication may occur at multiple layers within an agent system. At each progressive layer, communication is rooted on and abstracts that of the underlying layers.

the layers described in Section V-A, as shown in Fig. 13. Physical resources in the environment layer, such as cables, wireless signals, and network cards, allow devices in the host layer, the dashed ovals, to exchange network traffic. This is abstracted at the platform layer by operating system and network software as routing tables. At the framework layer, each platform may have one or more executing frameworks, denoted by smaller, dashed ovals. Typically, these instantiations may pass messages between each other, as shown by the lines between framework instantiations. Some agent systems may be equipped with *framework gateways* allowing the sharing of information between instantiations of different frameworks. In turn, these inter- and intraframework links provide for agents within framework instantiations to communicate.

## VI. Agent Framework Functional Concepts

This section presents a view of an agent system as a set of abstract functional concepts that support overall system execution. For example, *security* and *mobility* are two abstract functional concepts (among others) described. However, before beginning, two comments are in order. First, our use of the rather abstract term "concept" here is deliberate. The more concrete (and perhaps more familiar) term "component" could be used, but this term is somewhat misleading because a function often does not correspond directly to what engineers might think of as a component, i.e., a clearly delineated piece of the system. Instead, a functional concept is something that emerges out of complex interactions between pieces of software and hardware located in different layers of the agent system.

Second, this section makes few prescriptions about whether and how each functional concept is implemented. The way in which functional concepts are instantiated may vary significantly in structure, complexity, and sophistication across different agent system implementations. Indeed, some agent systems may not even possess some of the functional concepts described. The aim here is to describe what the function is in abstract terms so that one can determine if the function exists in a given system, or to verify its existence if it is claimed to exist within a given system.

### A. Agent Administration

*Definition:* Agent administration functionality: 1) facilitates and enables supervisory command and control of agents and/or agent populations and 2) allocates system resources to agents. Command and control involves instantiating agents, terminating agents, and inspecting agent state. Allocating system resources includes providing access control to CPUs, user interfaces, bandwidth resources, etc.

Agent administration functionality may be implemented in various ways. For example, the framework may perform all the administration functions directly, or there may be (multiple) agent(s) in the agent layer that perform agent administration functions by commanding and controlling other agents, or there may be elements of both approaches in a given system. For convenience of exposition later, the term "administrator" encapsulates all the administration functions although administration functions may not be necessarily implemented with a single administrator.

To further facilitate the exposition of the following process model, consider as an example a hypothetical system that uses agents to monitor message traffic on a communication network. The number of agents required to perform adequate monitoring may be contingent upon the complexity of the network topology or the priority of monitoring relative to other system goals. As both the network topology and priority changes, an administrator is employed to manage the network monitoring agents.

*Process model:* Agent administration functionality is described by the following set of processes.

1) *Agent creation:* It is the act of instantiating or causing the creation of agents. In the example before, the administrator may determine that there are too few network monitoring agents to adequately maintain a minimum level of security. Therefore, new network monitoring agents should be created.
2) *Agent management:* It is the process by which an agent is given an instruction or order. The instructions or orders could come from human operators, or from other agents. For example, if it is determined that the greatest security threat is over HTTP traffic, the administrator may request that the network monitoring agents focus their analysis on HTTP traffic.
3) *Resource control:* It is the process by which an agent's access to system resources is controlled. For example, the administrator may determine that security is of less priority than CPU usage. Therefore, it can reduce the available CPU time of the network monitoring agents.
4) *Agent termination:* It is the process by which agents are terminated (i.e., their execution is permanently halted). For example, the administrator might determine that there are too many network monitoring agents and decide to remove these in saturated regions of the network.

### B. Security and Survivability

*Definition:* The purpose of security functionality is to prevent execution of undesirable actions by entities from either within or outside the agent system while at the same time allowing

execution of desirable actions. The goal is for the system to be useful while remaining dependable in the face of malice, error, or accident.

*Process model:* Security functionality is described by the following processes.

1) *Authentication:* It is a process for identifying the entity requesting an action. Common examples include username/password credentials and use of public/private keys for digital signatures.
2) *Authorization:* It is a process for deciding whether the entity should be granted permission to perform the requested action. A common example in file system security is maintenance of a permission list for each file, which specifies the allowable actions for a given user. Another example includes a Web server denying a request to to view a page, due to the user whose credentials were used having insufficient permission.
3) *Enforcement:* It is a process or mechanism for preventing the entity from executing the requested action if authorization is denied, or for enabling such execution if authorization is granted. A common example for preventing access to information is to encrypt it. Permission to access the information is granted by providing the entity a decrypted copy or providing the entity the means to decrypt it, e.g., the encryption key.

Some general technologies for achieving security include authorization models and mechanisms, auditing and intrusion detection, cryptographic algorithms, protocols, services, and infrastructure, recovery and survivable operation, risk analysis, assurance including cryptanalysis and formal methods, and penetration technologies including viruses, Trojan horses, spoofing, sniffing, cracking, and covert channels.

### C. Mobility

*Definition:* Mobility functionality facilitates and enables migration of agents among framework instances typically, though not necessarily, on different hosts. The goal is for the system to utilize mobility to make the system more effective, efficient, and robust.

Mobility functionality is useful if, for example, the power level is low on a particular host and an agent may wish to migrate to another host to stay alive, or, an agent may need to communicate at length with an agent on another host, and so, it would be more bandwidth-efficient for the agent to migrate hosts rather than to send the communications over the network.

As shown in Fig. 14, mobility capabilities exist along three axes. The *mobile state* axis represents the capability of the state of execution (such as the instruction counter) to migrate with the agent. The *mobile code* axis represents the capability of code (byte code or platform specific) to migrate with the agent. The mobile computation axis represents the capability of the state of data members to migrate with the agent.

The four rounded corners in the figure represent how a framework can be classified, based on the mobility support it provides. *Process migration* refers to the mobile state and mobile computation support, *weak mobility* refers to mobile code and mobile

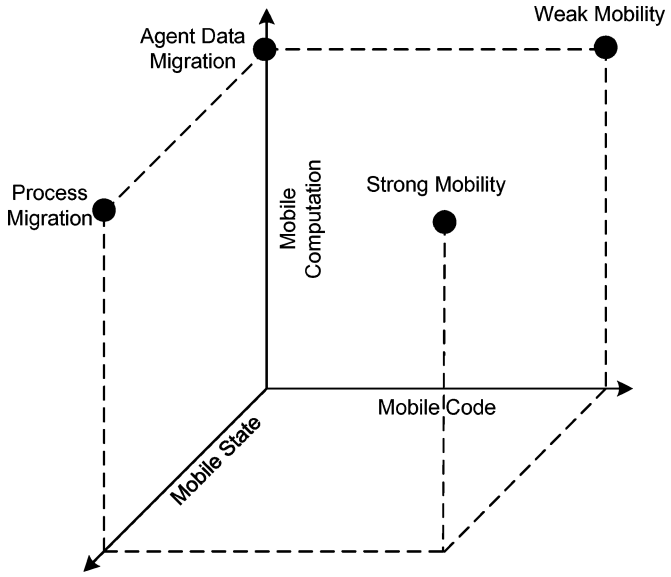


Fig. 14. Axis of mobility features adapted from [43].

computation support, and *strong mobility* refers to mobile code, mobile state, and mobile computation support. All classifications include mobile computation support. *Agent data migration* refers to support only for mobile computation support, and is the classification most common in contemporary agent frameworks.

*Process model:* Mobility functionality is described by the following processes.

1) *Decision procedure for migration:* It is a process for determining whether or not a migration should occur. The decision procedure can be *passive* or *active*. *Passive mobility* occurs when the decision to migrate is made outside the agent. For example, another agent, framework, host, or management service may determine when and where the agent shall migrate. An example of passive mobility is the mobility service provided by the Cougaar agent framework. By contrast, *active mobility* occurs when the agent is in control of its own mobility, and decides when and where it shall migrate on its own. An example of active mobility is the internal agent mobility functionality provided by Jade. In either case, it is decided that the agent shall migrate, and a suitable destination is chosen.
2) *Deregister, halt, and serialize:* Once an agent has decided (or been notified) that it is migrating, it must deregister from all of the directory services on the framework instantiation with which it has registered. Then, it halts execution, and is serialized.

   The serialization process involves persisting the agent's data and/or state into a data structure. This data structure is converted to packets or written to a buffer to prepare the agent for migration. In an object-oriented language, the data that must be stored are the data members of the object. Some frameworks may support storing other information, such as the point at which execution stopped.
3) *Migrate:* It is the process by which the serialized, nonexecuting agent leaves the source framework instance and arrives at a destination framework instance. This does not

necessarily imply that the agent has left the host; instead, the agent is changing the framework instance on which it is executing. Recall that a host and platform may be housing multiple framework instances, thus allowing for migration within a particular host. According to [44], mobility is also recognized as an atomic function. As a result, agents in a mobile state are not executing and cannot act until the agent resumes its behavior at the destination.

There is no requirement that an agent's destination framework instance is different from its source framework instance, i.e., an agent could serialize and "migrate" to itself. However, an agent system as a whole possesses mobility functionality if and only if it allows for agents to migrate among different framework instances.

4) *Deserialize, reregister, and resume:* Corollary to serialization is the process by which the agent, having arrived at its destination, is converted from its serialized state into the data structure that it existed as on the sending host. Then, the agent reregisters with the appropriate directory services in use by this framework and resumes execution. As noted in the mobility description, the agent can either resume execution where it stopped on the sending framework instantiation or restart from the beginning, depending on the support given by the framework.

Throughout the mobility process, exceptions can occur causing the mobility to fail. For example, during the migration process, the target host may refuse the agent, or network communication with the destination host may go down. Handling a failed migration is implementation specific. It is left to the system implementation to handle and recover from such exceptions.

### D. Conflict Management

*Definition:* Conflict management functionality facilitates and enables the management of interdependencies between agents activities and decisions. The goal is to avoid incoherent and incompatible activities, and system states in which resource contention or deadlock occur.

As an example, a framework may allow designation of superior/subordinate relationships between agents and provide generic conflict resolution services based on these relationships. The Cougaar framework does this. Similarly, a framework may provide a multiagent task planning language, such as TAEMS [45], that can be used to reason about the interactions between agent actions and to detect plan conflicts.

*Process model:* Conflict management functionality is described by the following processes.

1) *Conflict avoidance:* It is a process or mechanism for preventing conflicts. Examples of such processes include multiagent planning algorithms (both online and offline) that take care to produce action plans that do not have conflicts.
2) *Conflict detection:* It is the process of determining when a conflict is occurring or has occurred. One example includes a plan execution monitoring algorithm that is able to sense when the actions of agents are in conflict. Another example includes performing logical inference over different agents beliefs to determine when they are inconsistent with one another.
3) *Conflict resolution:* It is the process through which conflicts between agent activities are resolved. Negotiation, mediation, and arbitration are common mechanisms for facilitating conflict resolution.

Some general technologies for conflict management in agent systems include argumentation and negotiation, distributed constraint reasoning, game theory and mechanism design, multiagent planning, norms, social laws, and teamwork models.

### E. Messaging

*Definition:* Messaging functionality facilitates and enables information transfer among agents in the system.

This concept is associated specifically with the mechanisms and processes involved in exchanging information between *agents*. Although information exchange via messages can and often does occur between other parts of the system—for example, between an agent and its framework, between frameworks, between a host and its platform, etc.—such information transfer is not included because it is in a sense at a lower level. The concept of messaging used here is at a higher level than that associated with network traffic or interprocess communications.

Messaging involves a source, a channel, and a message. Optionally, a receiver may be designated, and models in which messages do not have a specific intended receiver are acceptable. For example, signaling in the environment like smoke signaling, a light flashing Morse code, etc., are examples of messaging where there is no designated receiver. Many other functional concepts such as conflict management and logging may utilize messaging as a primitive building block. Other functionality in support of concepts such as semantic interoperability and resource management may be necessary to practically or effectively conduct messaging. However, messaging is defined here as a stand-alone concept of its own right.

*Process model:* The functionality is described by the following processes.

1) *Message construction:* It is the process through which a message is created, once a source agent determines its wish to deliver a particular message chosen from a finite or infinite set of messages. No commitments are made here in regard to the form, structure, or content of a message. For the purposes of this model, it is sufficient to discuss messages as an abstract object. The information to be delivered is simply the fact that a particular message was chosen from the set of all possible messages.
2) *Naming and addressing:* It is a mechanism for labeling the message with its intended destination or route. Directory white page services are a common mechanism to facilitate this function. Broadcast, multicast, and group messaging also all fit within this model.
3) *Transmission:* I is the actual transport of the message over the channel. This may be a one-shot transmission or a continuous stream. One common model of interhost agent messaging involves going through the platform to the host's network hardware, then out into the environment (via wire or air), and back in symmetrically to the receiver.

4) *Receiving:* It is the process for acquiring the transmitted information so that is usable by the receiver. This may be as simple as pulling the message off of a queue or more elaborate, e.g., going through a translator.

Some other areas of interest in messaging functionality include notions of best effort delivery, quality of service (QoS), and guaranteed delivery/timeliness.

### F. Logging

*Definition:* Logging functionality facilitates and enables information about events that occur during agent system execution to be retained for subsequent inspection. This includes but does not imply persistent long-term storage.

Logging is a supporting service that provides informational, debugging, or management information about the agent system as it executes. It can be a centralized service or distributed among the agents (wherein each agent performs its own logging). Logging services are often used to make note of system-wide information or warnings produced by the agent or the agent system.

*Process model:* Logging functionality is described by the following processes.

1) *Log entry generation:* It is the process by which information to be logged is created. For example, a log entry may be a note of immediate importance regarding the system: for instance, a damaged sensor or low battery life. The entry could be generated whenever an agent has entered a particular state or generated regularly to aid system status monitoring. While these entries have different meanings and priorities, they can be generated in the same manner. Log entries often include type (informational, warning, critical, among others) or priority (for instance, priority 1–5). The entry and any attributes are packaged into a data structure for writing.
2) *Storing log entry:* Log entries are stored in a variety of ways at the choosing of the implementation of the agent system. For example, log entries can be written to a disk file on a host, written to a network stream destined for another agent, simply stored in memory for debugging purposes, or written to a generic stream with a defined destination. The log message is optionally formatted, often into a textual description or a database format such as XML.
3) *Accessing log entry:* The logging functionality must provide a mechanism for a human user or an agent to access the generated log entries. If the entry contained any attributes, such as priority or type, they are also accessible. For example, if the agent is in a critical state, an agent system management service or human intervention may be alerted to this by accessing the log information. A log filter may also be available for facilitating listing and reading the log entries.

### G. Directory Services

*Definition:* Directory services functionality facilitates and enables locating and accessing of shared resources.

A directory is an abstraction allowing the naming and registration of resources enabling subsequent locating of and access to the resources. Examples of shared resources located and accessed through a directory service include other agents or services. Directory services are often used to locate agents and services with specific characteristics.

*Process model:* Directory services functionality is described by the following processes.

1) *Naming:* It is the process by which resources are assigned identifiers so that they may be indexed and located. This process can be fairly complex by supporting group names, transport addresses, dynamic name resolution, and other complex features [46].
2) *Notification:* It is the process by which new resources are added to and deleted from the directory. As resources dynamically become available and unavailable, the directory is kept up-to-date via this notification process to maintain an accurate picture of the resources available in the system. When a new resource is added, the process often includes recording a description or characteristics of the resource and a method for accessing it.
3) *Query matching:* It is the process by which resources are looked up in the directory. This process often occurs in response to external requests for a resource and returns information about how to access the requested resource. Queries can be specified in terms of the name of the service (e.g., white pages directory) or by a service description (e.g., yellow pages directory) [47].

## VII. Using the ASRM

Given the aforementioned terminology and definitions, we now show the practicality of the reference model by using it to compare agent frameworks and examine deployed systems built with agent technology.

### A. Agent Framework Mappings to the Reference Model

Ideally, all existing agent frameworks map directly to the reference model from an architectural perspective. Because of the number of diverse frameworks in existence, each with its own functional goals and architecture, it is not feasible to compare all of the existing frameworks to the reference model; however, an analysis of a representative subset is presented in this section. The following is the general behavior that is used to exercise the individual frameworks. Any modifications are noted appropriately.

> Two static agents $s_1$ and $s_2$ are created. One of the static agents $s_1$ creates a mobile agent $m$. It is the responsibility of $m$ to deliver a message from $s_1$ to $s_2$. Mobile agent $m$ then migrates from the framework instantiation on which $s_1$ resides to the framework instantiation in which $s_2$ exists. Once the message is delivered, $m$ returns to $s_1$ and $s_2$ is terminated. Upon arriving back at $s_1$, $m$ is terminated, and finally, $s_1$ is terminated.

This behavior tests the migration and message passing aspects of several agent frameworks. Typically, these components also exercise the other components described in the reference model. For example, migration requires a search of the

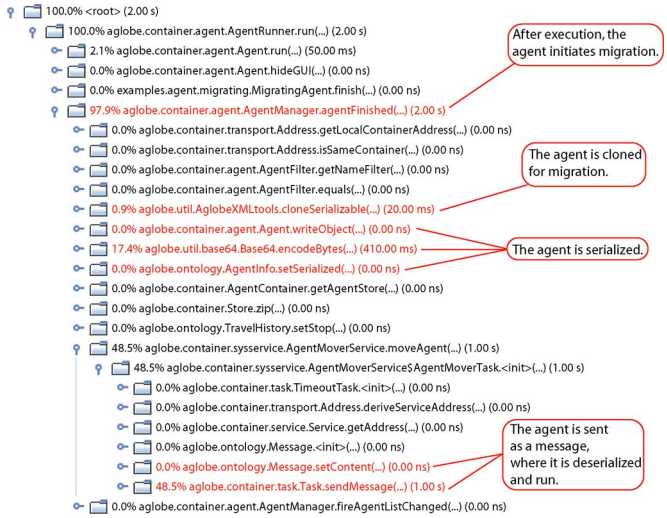


Fig. 15. A-Globe migrating agent $m$ before migration. A-Globe agents migrate using a procedure consistent with the ASRM mobility functional concept.

directory service, has security concerns, needs to deal with agent management functions, and involves coordination. Likewise, the message passing aspect generally exercises communications and security. A brief overview of the scenario from a dynamic analysis is first presented, followed by in-depth analysis highlighting components and tracing execution.

Execution of this behavior is traced using the Extensible Java Profiler (EJP) tool. From these traces, one draws conclusions about the framework's architecture and makes mappings to a reference architecture (and thus to the reference model). Several figures are included in the next few analysis sections. These figures show what amounts to the raw output of EJP. The runtime trace is clearly recognized in the tree structure depicted. Every node represents a function call that was performed by the parent. The percentage of total time spent executing a particular call is shown as a percentage after the function name. Some of the calls are removed for the sake of clarity. These include the standard Java library calls. Lastly, equivalent and consecutive calls are commonly grouped together as a single method invocation.

*1) A-Globe:* This A-Globe analysis is broken up into two behaviors for depth of analysis. One part involved message passing and the other involved migration—thus deviating from the general scenario. Combining these results in the generic behavior depicted for framework analysis.

*a) Overview:* An analysis of an agent frameworks begins with the instantiation of the framework itself. A-Globe uses the platform class as the root of its framework. The platform class controls containers that are for all intents and purposes agents. The `AgentContainer` acts as the interface between local agents and the framework instantiation. Its main job is to provide agent-specific resources such as a `MessageTransport` services. An `AgentManager` is used to manage the agents and is seen in the following figures for the specific agents.

The migrating agent before migration is seen in Fig. 15. The general procedure for a migrating agent is to run (in this case, the migrating agent does nothing), and then migrate using the `agentFinished` function. It migrates to the second instantiation of the agent framework. Again, the agent does nothing and is terminated by the agent framework instantiation in the `agentFinished` function.

*b) Mapping to agent framework functional concepts:* An examination of the A-Globe [9] figures reveals the manner in which the architecture of A-Globe maps to the idealized agent framework. First and foremost, the physical world and infrastructure are implicit. The Java Virtual Machine is the only visible sign of a platform. It occupies the root nodes through any thread instantiations in the trees.

The framework is represented by the platform class and its corresponding `AgentContainer`. The framework is always used for the agent to access system or physical world resources. The platform's agent container provides shared objects, a message transport service, a directory service, a logger (not shown), and various other resources to the agents. An `AgentManager` manages the agents on the local instantiation of the framework. It takes care of the migration aspect.

To begin, consider the migrating agent. It was not implemented to perform any task on the local host besides migration, so the run method does very little. When it is time to migrate, the `agentFinished` function is called. This function uses XML to serialize the data and sends it via the `AgentMoverService` that is part of the `AgentContainer`. This is shown in Fig. 15. When the migration agent arrives at the second framework instantiation, it is recreated by the `AgentManager` and executed. Again, it does nothing and is then deregistered from the directory terminated.

There is evidence that there exists a `MessageTransport` service within the framework that oversees message passing. This `MessageTransport` service registers possible message receivers, and then calls a function similar to the `handleMessage` function.

Overall, migration and message passing are the same as in the idealized agent framework. In a combination of these experiments, $s_1$ generates a message and gives it to the framework to give to $m$. Migrating agent $m$ then receives the message and migrates to the instantiation of the framework where $s_2$ resides. After arriving, $m$ passes the message along to $s_2$. Finally, all framework instantiations and agents are terminated. The framework also makes available all of the important components mentioned in the reference model in addition to the migration, communication, and logging components exercised in these experiments.

*2) Jade:* The Jade [11] agents implement the messaging and migration scenario by using a `TerminatorAgent` that is responsible for creating agents and terminating the platform upon completion. The `TerminatorAgent` creates two static agents on one host that send a message to each other. Thus, there exists $s_1$ and $s_2$ on the first host. The `TerminatorAgent` is the mobile agent $m$, and travels to the second host where it repeats same task on new static agents $s_3$ and $s_4$. These two new agents serve the same purpose as the original $s_1$ and $s_2$ agents; the entire task is repeated in order to analyze the interactions between the `TerminatorAgent` and the agent management subsystem


Fig. 16. Dynamic analysis data for Jade static agents. Here, the agent is initialized and run by the framework.

when creating and terminating agents. This behavior is discussed elsewhere in the ASRM. We chose to do this because it yields data on several ASRM functional concepts during the same execution trace, which enables us to inspect relationships and dependencies that exist between functional concepts in each agent framework.

*a) Overview:* The Jade framework is composed of several classes, including `Boot` and the entire `jade.core` package. The `AgentContainer` is used as an interface between the agents and the framework. Additionally, resources are delegated through the framework by way of a `ServiceManager`. The static agents are very interesting in the case of Jade because the progression of message sending is made obvious. All of the static agents are the same and appear as in Fig. 16.

*b) Mapping to agent framework functional concepts:* In Jade, the environment and host layers are implicit while the Java Virtual Machine (JVM) represents a portion of the platform. When the Jade framework is started, an `AgentContainer` is created that interfaces the framework with all local agents. Inside the `AgentContainer`, the other agents are started. The Jade framework also contains an `AgentManagementService`, a `MessagingService` with a well-defined and FIPA compliant Agent Communications Language (ACL), an `AgentMobilityService` class that oversees mobility, and a `ServiceManager` that manages local resources. A directory service is hidden in the `jade.domain.ams` package.

First, message passing is examined in detail. Two `JadeCommunicationAgents` are created that send messages to each other, and then terminate. In Fig. 16, the static agents take advantage of Jade's planning language that includes behaviors that are defined to occur once or be cyclic. The agent sends a message by creating an `ACLMessage`, contacting the directory service with the `getAMS` function call, and using the communication service to send. This is a `OneShotBehaviour` and occurs only once per lifetime. A `CyclicBehaviour` attempts to receive messages and terminate, but is left unexpanded in the

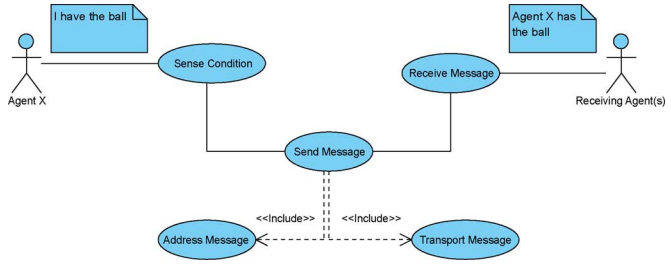


Fig. 17. Robot messaging use case.

figure. Thus, message sending and receiving are made clear in the static agent figure.

The migrating agent exhibits some interesting properties both before and after migration. The agent creates the static agents, and then attempts to clone itself. Migration is processed through the `AgentMobilityService`. The directory service is certainly contacted in this process. The first function call regenerates the agent and it proceeds in a similar fashion by creating the communication agents.

Jade contains a framework consisting of all of the ASRM components.[5] Agents are given the freedom to operate autonomously, but are monitored through an `AgentManager`. The other components are also present explicitly; security is the only exception that is not obvious in the dynamic analysis, though it is implicit through the JVM and the API.

## B. Case Studies

*1) Situated Agent Example—Robot Soccer:* Agents are typically situated within an environment and are able to interact among themselves and with that environment through its appropriate framework and infrastructure. An example is a model of agents represented by robots, whose goal it is to play soccer in a league. The agents in this case vary by size and complexity, and may have storage constraints based on the rules of the particular league. This example is an analysis of the potential behavioral interactions of situated agents such as robots in the context of the ASRM concepts and agent system layers.

*a) Messaging scenario:* A typical agent messaging scenario in the robot soccer domain is a situation in which an agent senses a condition that merits communicating. In this example, the agent senses that it has successfully acquired the ball. This is critical information to pass along to the team to inform them of the condition and begin the planning process of making a play. In addition, it is possible that some agents have lost sight of the ball and may have become disoriented on the field; an informative message from a teammate is also helpful in this situation. In either case, it is possible to both send a message to a single agent or to a set of agents (see Figs. 17–19).

*b) Sending agent:* The sending agent senses a condition that merits sending a message. In this case, the agent X "has the ball." To this effect, the agent decides to send a message to its teammates.

[5]However, the security and mobility components cannot be employed concurrently.

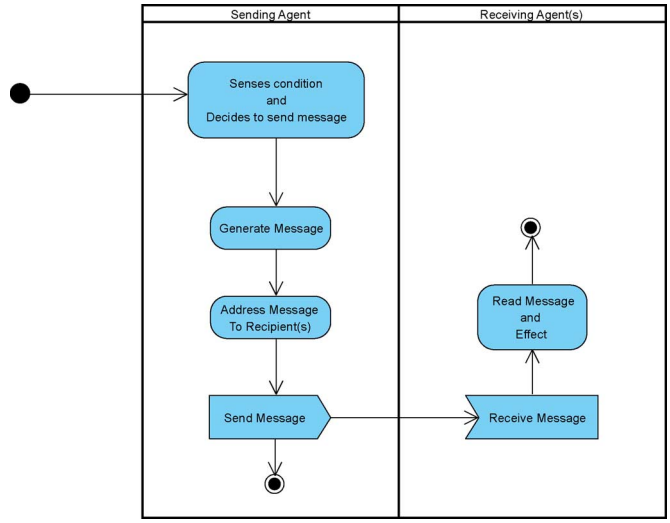


Fig. 18. Messaging activity diagram.

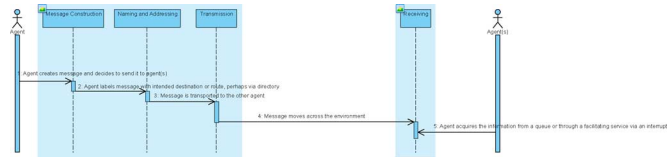

Fig. 19. Messaging sequence diagram.

*c) Sending framework:* The sending framework API is called for sending a message. The message body is constructed and wrapped in an envelope that contains logistical information such as addressing. In this case, the body of the message is "agent X has the ball" and the address is that of each of agent X's teammates.

*d) Sending platform:* The operating system and network driver receive a low-level request to build a message that is identical to that mentioned previously. The message is broken up into packets that are sent individually.

*e) Sending host:* The network card physically transmits the packets through the network pins or over the wireless antenna.

*f) Environment:* The environment in this scenario is the actual network. The packets are transmitted over the medium.

At this point, the process is reversed and the message is received.

*g) Receiving host(s):* The network card physically receives the packets from the network or via the wireless antenna.

*h) Receiving platform(s):* The packets are received by the operating system and network driver, and the packets are combined into a low-level message.

*i) Receiving framework(s):* The framework call is initiated by the agent and contains an API to check with the network interface to determine if a message is incoming. This could also be achieved via an interrupt.

*j) Receiving agent(s):* The agent receives the message via the framework API call and interprets it according to its decision cycles. In this case, the agent learns that agent X "has the ball."

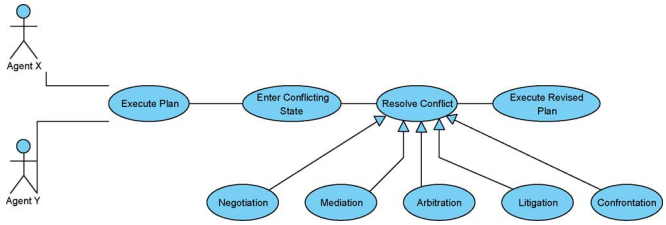


Fig. 20. Robot conflict management use case.

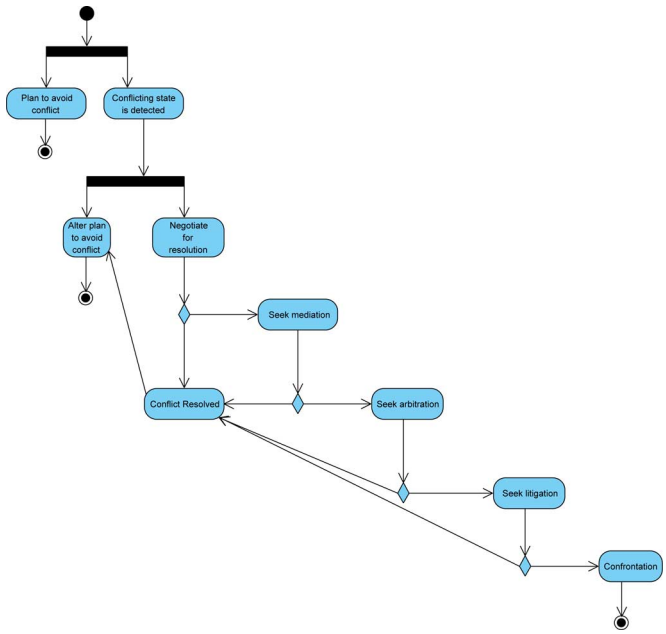


Fig. 21. Conflict management activity diagram.

*k) Conflict management scenario:* There are a number of possible scenarios even in the context of robot soccer where conflict resolution becomes necessary. Moreover, there are a number of ways to resolve such a conflict, and the appropriate method is usually a decision made by the agents as part of their plan and decision cycles.

Particularly, conflict resolution falls into two categories: centralized and decentralized. In centralized conflict resolution, there exists a management agent (coach in this context) that can "see" the entire field and has a plan for the entire team. In this case, the manager simply makes a decision and dictates/suggests the next actions for the team. More often, however, agent conflict resolution is decentralized and it is up to the agents in conflict to come to a resolution.

From the command and control context, the process of conflict resolution is broken into escalating stages: negotiation, mediation, arbitration, litigation, and confrontation. This scenario concentrates on the first three stages as an ongoing process.

In this scenario, two teammates are "fighting" over possession of the ball. Clearly, this is detrimental to their own goals, and therefore, a decision must be quickly made in a decentralized way. It is assumed that there is no "coach" to help them, but if there was, the resolution would be a decision process by the coach followed by message passing to the agents in conflict (see Figs. 20–22).

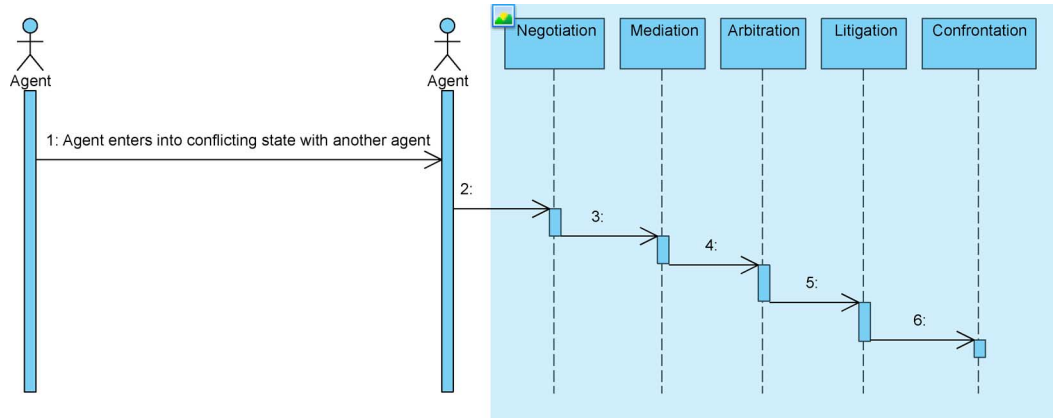


Fig. 22. Conflict management sequence diagram.

In this situation, the two agents have entered a conflicting state because they are both fighting for the ball. If the two agents are teammates, then they share a common goal. In this case, it is unlikely that the agents wish to continue "fighting" for the ball or even to seek arbitration to resolve the conflict. Instead, the two agents negotiate or collaborate with their confidence levels of achieving that goal. If one agent feels more likely to achieve the goal, then the other agent backs off and allows the more confident agent to take the ball. This confidence might be a confidence to score, a higher battery life, or an ability to run faster on the field. In any case, the other agent likely replans and positions itself in such a way as to assist the more confident agent, because they do indeed share a common goal.

Should negotiation result in indecision, and each agent has an equal confidence level, then the conflict must be resolved by the team or by a coach. This could be done via a group decision such as voting, followed by message passing to the agents in question.

If, however, the two agents have conflicting goals and are opponents, a different procedure is necessary. In fact, there is no negotiation or mediation in this scenario. Instead, either a referee needs to call the play dead (a form of arbitration), or the two agents are left to fight for the ball.

*l) Security and survivability scenario:* Because of the rules of the robot soccer league, security is enforced among the agents. For example, messaging occurs over discrete frequencies, and it is not permitted for agents to "eavesdrop" on opponents' communications nor to tamper with them. It is not legal for a robot to "pose" as a teammate of its opponent, and so on. Therefore, security is not realistic in this model and is appropriately omitted.

However, in similar scenarios, these assumptions are clearly not appropriate. For example, if robots are fighting on a battlefield, security could not be imposed by a league as rules. The security concepts of authentication, authorization, and enforcement (encryption, etc.) would be exercised during every agent interaction. In this way, security functionality can be "plugged in" to many of the other use cases shown here.

As in other examples, the agent begins by deciding that it wishes to execute a particular plan or functionality; for example,

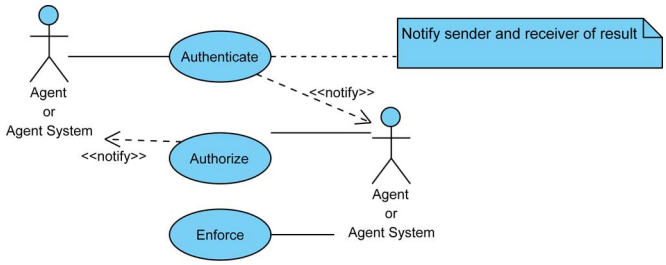


Fig. 23. Security use case.

it wishes to communicate with a group of agents or migrate to another host. This functionality is protected through some controlling authority, which could be a group of agents acting in a decentralized way (through voting, for example) or through a more centralized system such as a certificate authority (CA).

However, the agent authenticates itself with the controlling authority by sharing credentials such as a username and a password. This information is validated and the authentication stage completed. If authentication is successful, an agent then enters the authorization state, in which the controlling authority determines if the agent has the appropriate permissions or status to execute the desired functionality. If so, authentication passes and the agent is allowed to execute. If not, the policy is enforced and the agent is unable to execute. This can be achieved in a number of ways, and is often affected through the use of encryption keys (see Fig. 23). This process is illustrated in more detail in Figs. 24 and 25.

*m) Mobility scenario:* Similarly, mobile code is not often found in the robot soccer domain, but is certainly central to many situated agent systems. In general, agents migrate by serializing their state that is subsequently transported to another agent framework instance, as defined by the reference model and illustrated in Fig. 26.

At an activity level, an agent uses its sensor and effector interfaces to determine that migration is necessary and feasible. This could be facilitated through a mediating party or independently through the agent's own decision cycles. Once the decision has been made to migrate, the agent is serialized by the framework,

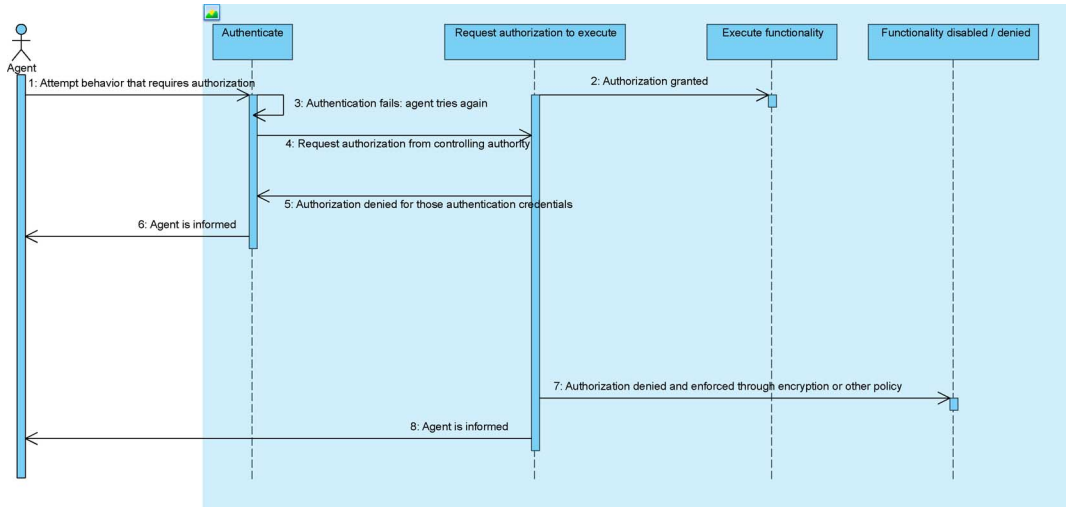


Fig. 24. Security sequence diagram.

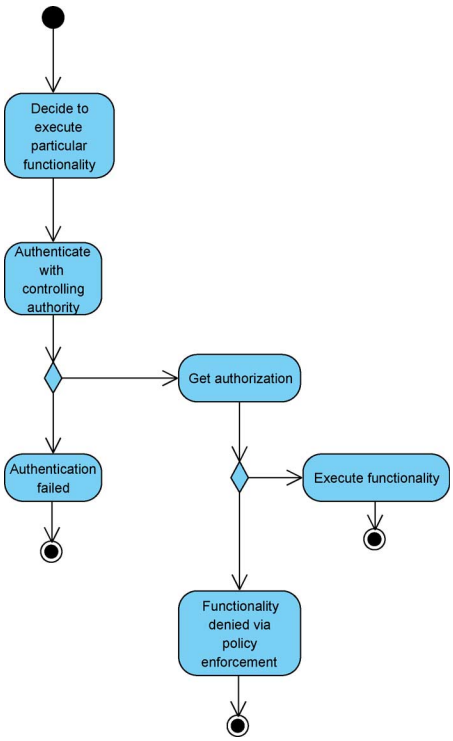


Fig. 25. Security activity diagram.

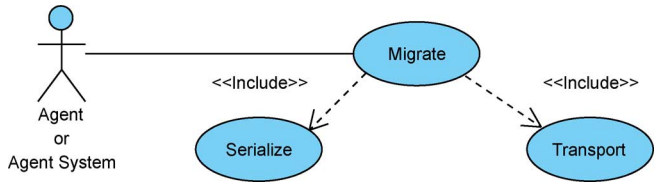


Fig. 26. Mobility use case.

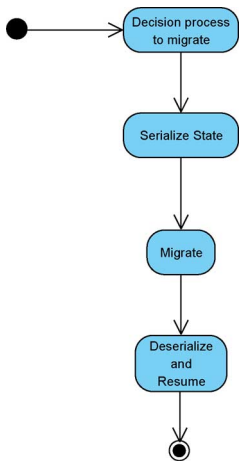


Fig. 27. Mobility activity diagram.

transported through the network as appropriate, and deserialized at the destination. This is illustrated in Fig. 27 and described in detail in Fig. 28.

*2) Situated Agent Example—Secure Wireless Agent Testbed (SWAT):* The SWAT [48] is a unique facility to study integration, networking, and information assurance for next-generation wireless mobile agent systems. It integrates:

1) mobile agents;
2) multihop, mobile *ad hoc* wireless networks (MANETs);
3) security and information assurance.

SWAT's agent-based applications are implemented in Java and have been tested with the EMAA [10] agent framework. This section details how it implements the core components of the ASRM.

*a) Messaging:* Messaging services are provided by the EMAA framework. These services are used to support communications among application agents that provide end-user tools such as a whiteboard, voice over IP (VoIP), GPS tracking, and other components.

*b) Scenario:* In order to accomplish the distribution of real-time GPS data, each node that receives GPS input then securely sends this information to all other hosts on the network.

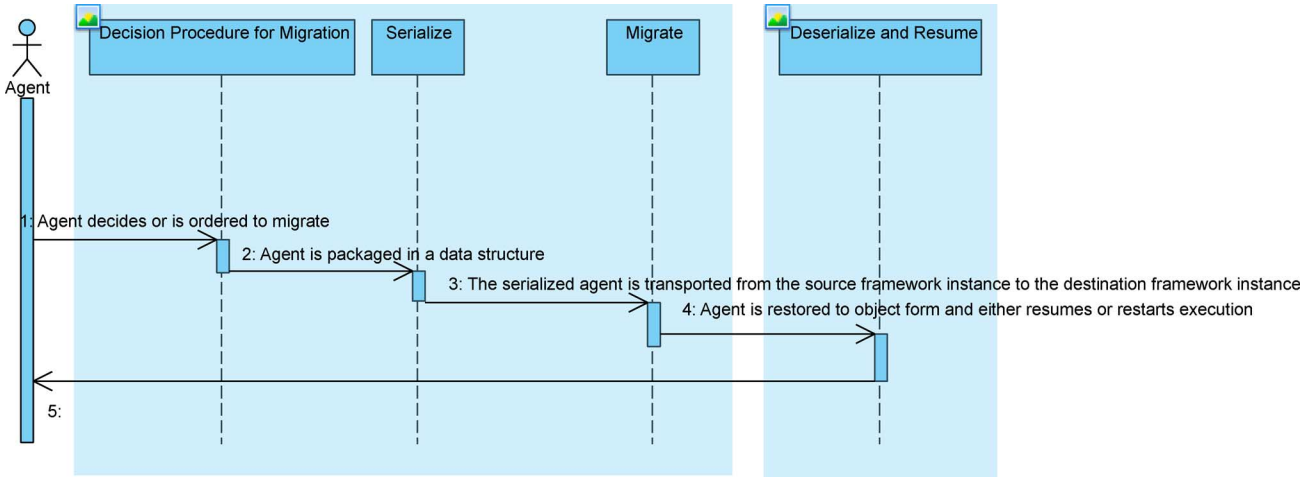


Fig. 28. Agent mobility and migration sequence diagram.

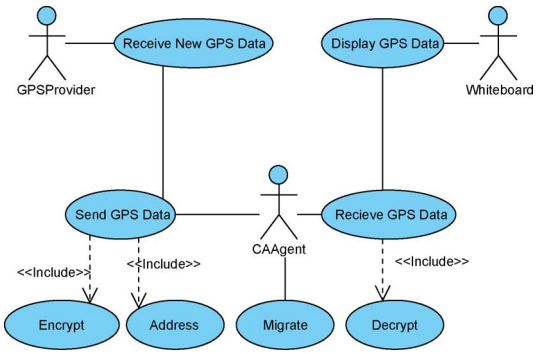


Fig. 29. SWAT messaging use case diagram.

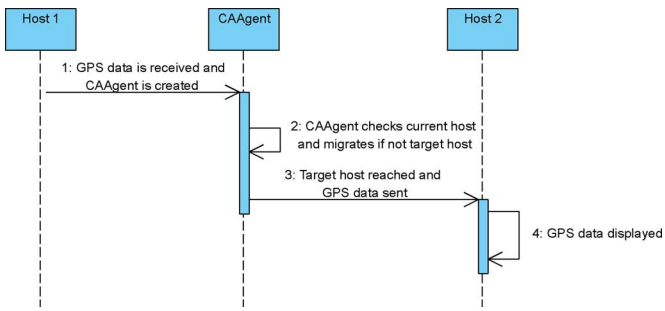


Fig. 30. SWAT messaging sequence diagram.

In order to minimize network traffic, a `GPSProvider` agent uses a synchronized hashtable containing hostnames and the status of the last message sent. The `GPSProvider` encrypts the data before creating a `CAAgent` to convey the data by migrating to the target host, as shown in Fig. 29. The messaging process is outlined in Fig. 30.

Once the GPS data reach the remote host, then that host decides, based on the whiteboard parameters, how to display the data.

*c) Mobility:* Agent mobility is handled strictly through the EMAA framework services.

*d) Scenario:* In the course of transporting GPS data, the `CAAgent` must migrate from one host to another. On a MANET, this implies a constantly shifting network topology. In order to maximize the agent efficiency, the `CAAgent` rechecks its path at every host it passes through. Mobility is detailed in Fig. 31.

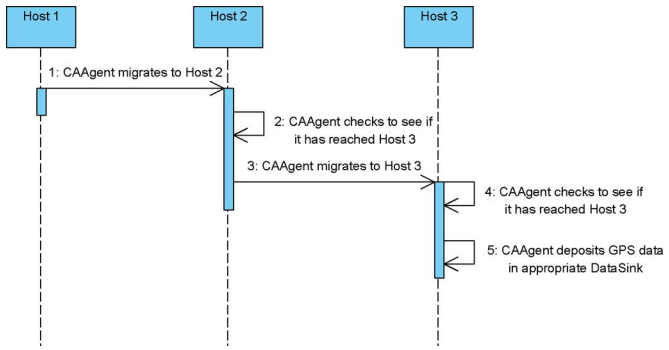


Fig. 31. SWAT mobility sequence diagram.

*e) Security:* The security manager details group membership management, as well as a directory service (agent lookup, group membership lookup, etc.). This security manager also handles all of the encryption/decryption services. There is also a CA that gives out private keys and a security mediator (SEM) that is used for revocation. In short, each host only has half of the necessary private key, so it must contact the SEM for the other half. The SEM responds unless the host was revoked. In such a way, messages can still be sent through revoked hosts because they can pass them along without being able to decrypt them.

*f) Resource management:* SWAT includes a service registry that distributes and lists available resources. This registry can be made global such that agents can query the central registry to determine where a particular resource is located. Items in the registry can be looked up by name or description.

SWAT implements these services directly, as they are not provided explicitly in the agent framework itself.

*g) Scenario:* In the course of delivering GPS data, a `CAAgent` must query the service registry to find the proper target of its data once it reaches the target host. Therefore, when it is started, the whiteboard application's `GPSOverlayPlugin`

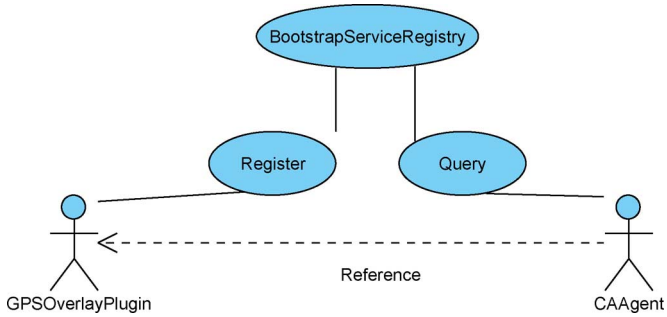


Fig. 32. SWAT resource management use case diagram.

registers itself as a `GPSSink`. This stores a reference to the `GPSOverlayPlugin` in the service registry, which can be used to call specific methods of that class (see Fig. 32).

*h) Group management:* SWAT makes use of the Spread toolkit [49] for communication services for security and group management. In this context, it is a separate communications channel from EMAA's agent-to-agent messaging. Spread is a fault-tolerant messaging service. As mentioned in the Section VII-B2(e), a CA handles most group management functions by allowing and disallowing members in the global group (i.e., all those on the wireless network) to join a specific group. Each group has a unique public and private key securing intragroup communication. Group members can share many sorts of data: whiteboard annotations, VoIP messages, and GPS information. Due to security, intergroup communication is not possible except with the CA. Any major event, such as a join or a leave, spawns a rekeying sequence.

*i) Routing:* SWAT can use any of several MANET protocols to create network routes. These protocols are separate from the agents; however, they are necessary for agent-to-agent communication. The optimized link state routing [50] protocol, for example, supports a periodic survey of network state and the sharing of route tables with neighbors. Consequently, each host has a relatively current global topology of the network at any given time. When messages are sent, the shortest path in numbers of hops is the primary factor in determining a route; other factors, such as link quality, are secondary.

## VIII. Conclusion

This paper introduced the ASRM [1] and provided a set of detailed examples of how the ASRM can be used to better understand and integrate agent-based systems. In the course of developing the ASRM, dozens of agent frameworks were analyzed using software analysis techniques. The data that resulted from techniques provided a detailed map of how existing software developers design the frameworks for the creation and execution of agents. The ASRM takes no stand on what might make an agent "intelligent," leaving that to the developer to implement as part of the control mechanisms of the individual agents. The ASRM's focus is on enabling system-level descriptions of how agents (potentially from different frameworks) interact. To this end, the ASRM has used evidence from the software analysis results to define a five-layered model for how agents are situated in the world and interconnected as part of an agent-based system. The analysis results also provide information regarding minimum conformance requirements for each of the layers as well as the process models for functional components (i.e., security, management, etc.) in the generic agent framework. To support the utility of the ASRM, we provided several examples of how the ASRM can be used to better understand existing agent systems. It is the belief of the authors that the ASRM fills an important missing component in agent systems research. Completely compatible with FIPA, the ASRM enables developers to map where FIPA and other standards are best used in the context of a large, heterogeneous agent-based system. Further, the ASRM provides the beginnings of a roadmap for agent framework developers to guide system development issues and make rational decisions about which system layer is best for providing needed functionality to agents. Every attempt has been made to make this paper's contents highly clinical and free of the subjective opinions that sometimes dominate agent literature. It is our hope that the ASRM stimulates additional discussion and enables researchers to advance the utility and application of agent technology to emerging problems of global need.

## Acknowledgment

The authors would like to thank the members of the U.S. Army Intelligent Agents SubIPT[6] for their input on this paper.

## References

[1] I. Mayk and W. C. Regli, Eds. (2006, Nov.). *Agent Systems Reference Model*. Intelligent Agents Integrated Product Sub-Team, Networking Integrated Product Team, Command and Control Directorate, Headquarters, US Army Research, Development, and Engineering Command, Communications-Electronics Research, Development, and Engineering Center, Department of the Army [Online]. Available: http://gicl.cs.drexel.edu/people/regli/reference_model-v1a.pdf

[2] R. Malveau and T. J. Mowbray, *Software Architect Bootcamp*. Englewood Cliffs, NJ: Prentice-Hall, 2001.

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

[4] D. B. Lange, M. Oshima, G. Karjoth, and K. Kosaka, "Aglets: Programming mobile agents in java," in *Proc. Int. Conf. Worldwide Comput. Appl.*, 1997, pp. 253–266.

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

[6] K. Sycara and A. S. Pannu. (1998, May 9–13). The RETSINA multiagent system: Towards integrating planning, execution and information gathering. *Proc. 2nd Int. Conf. Auton. Agents (Agents 1998)*, K. P. Sycara and M. Wooldridge, Eds. New York: ACM Press [Online]. pp. 350–351. Available: http://www.acm.org/pubs/articles/proceedings/ai/280765/p350-sycara/p350 -sycara.pdf

[7] S. Poslad, P. Buckle, and R. Hadingham, "The FIPA-OS agent platform: Open source for open standards," in *Proc. 5th Int. Conf. Exhib. Practical Appl. Intell. Agents Multi-Agents*, 2000, pp. 355–368.

[8] C. Baumer, M. Breugst, S. Choy, and T. Magedanz, "Grasshopper—A universal agent platform based on OMG MASIF and FIPA standards," *Proc. 1st Int. Workshop Mobile Agents Telecommun. Appl.*, 1999, pp. 1–18.

[9] D. Sislak, M. Rollo, and M. Pechoucek, "A-globe: Agent platform with inaccessibility and mobility support," *Cooperative Inf. Agents*, vol. VIII, pp. 199–214, 2004.

[6]Integrated Product/Process Team.

[10] R. P. Lentini, G. P. Rao, J. N. Thies, and J. Kay, "EMAA: An extendable mobile agent architecture," in *Proc. AAAI Workshop Softw. Tools Dev. Agents*. Madison, WI: AAAI, Jul. 1998.
[11] F. Bellifemine, A. Poggi, and G. Rimassa. (1999). JADE–A FIPA-compliant agent framework. *Proc. 4th Int. Conf. Practical Appl. Agents Multi-Agent Syst. (PAAM 1999).* London U.K.: The Practical Appl. Company Ltd. [Online]. pp. 97–108. Available: http://sharon.cselt.it/projects/jade/PAAM.pdf 2009.
[12] N. Suri, J. M. Bradshaw, M. R. Breedy, P. T. Groth, G. A. Hill, R. Jeffers, T. S. Mitrovich, B. R. Pouliot, and D. S. Smith, "NOMADS: Toward a strong and safe mobile agent system," in *Proc. 4th Int. Conf. Auton. Agents*, C. Sierra, M. Gini, and J. S. Rosenschein, Eds. Barcelona, Spain: ACM Press, Jun. 2000, pp. 163–164 (Poster announcement).
[13] G. I. Inc. (2006). CoABS Web site [Online]. Available: http://coabs.globalinfotek.com/
[14] A. R. Silva, A. Romao, D. Deugo, and M. M. D. Silva, "Towards a reference model for surveying mobile agent systems," *Auton. Agents Multi-Agent Syst.*, vol. 4, pp. 187–231, 2001.
[15] *FIPA Abstract Architecture Specification*, Foundation for Intelligent Physical Agents, Dec. 2003.
[16] D. Chess, C. Harrison, and A. Kershenbaum, "Mobile agents: Are they a good idea?" International Business Machines Corporation (IBM), Yorktown Heights, NY, Tech. Rep. RC-19887, Dec. 1994.
[17] O. Etzioni and D. S. Weld. (1995, Aug.). Intelligent agents on the Internet: Fact, fiction, and forecast. *IEEE Expert* [Online]. *10(4)*, pp. 44–49. Available: ftp://ftp.cs.washington.edu/pub/ai/ieee-expert.ps.Z.
[18] C. Ghezzi and G. Vigna, "Mobile code paradigms and technologies: A case study," presented at the 1st Int. Workshop Mobile Agents, Berlin, Germany, Apr. 1997.
[19] A. Fuggetta, G. P. Picco, and G. Vigna. (1998, May). Understanding code mobility. *IEEE Trans. Softw. Eng.* [Online]. *24(5),* pp. 342–361. Available: http://www.elet.polimi.it/Users/DEI/Sections/Compeng/GianPietro.Picco/papers/tse98.ps.gz
[20] D. Kotz, R. Gray, and D. Rus, "Future directions for mobile-agent research," Dartmouth College, Hanover, NH, Tech. Rep. TR-2002-415, Jan. 2002.
[21] M. N. Huhns and M. P. Singh, Eds., *Readings in Agents*. San Francisco, CA: Morgan Kaufman, 1998.
[22] M. N. Huhns and M. P. Singh, "A multiagent treatment of agenthood," *Appl. Artif. Intell.*, vol. 13, no. 1–2, pp. 3–10, Jan.–Mar. 1999.
[23] R. S. Gray, D. Kotz, G. Cybenko, and D. Rus, "Mobile agents: Motivations and state-of-the-art systems," Dartmouth College, Hanover, NH, Tech. Rep. TR2000-365, 2000.
[24] M. Wooldridge, *Reasoning About Rational Agents*. Boston, MA: MIT Press, 2000.
[25] S. Hanks, M. E. Pollack, and P. R. Cohen, "Benchmarks, testbeds, controlled experimentation, and the design of agent architectures," *AI Mag.*, vol. 14, no. 4, pp. 17–42, 1993.
[26] S. P. Fonseca, M. L. Griss, and R. Letsinger, "Agent behavior architectures a MAS framework comparison," in *Proc. 1st Int. Joint Conf. Auton. Agents Multiagent Syst. (AAMAS 2002),* M. Gini, T. Ishida, C. Castelfranchi, and W. L. Johnson, Eds. Barcelona, Spain: ACM Press, Jul., pp. 86–87.
[27] H. S. Nwana, "Software agents: An overview," *Knowl. Eng. Rev.*, vol. 11, no. 3, pp. 1–40, Sep. 1996.
[28] L. Hagen, M. Breugst, and T. Magedanz. (1998, Aug.). Impacts of mobile agent technology on mobile communications system evolution. *IEEE Pers. Commun.* [Online]. *5(4)*, pp. 56–69. Available: http://www.ikv.de/download/grasshopper/IEEE-PCM-98.pdf
[29] K. P. Sycara, "Multiagent systems," *AI Mag.*, vol. 19, no. 2, pp. 79–92, 1998.
[30] N. R. Jennings, K. Sycara, and M. Woolridge, "A roadmap of agent research and development," *J. Auton. Agents Multi-Agent Syst.*, vol. 1, no. 1, pp. 7–38, 1998.
[31] G. Vigna, Ed., *Mobile Agents and Securit* (Lecture Notes in Computer Science). New York: Springer-Verlag, 1999.
[32] M. D'Inverno, *Understanding Agent Systems*. New York: Springer-Verlag, 2001.
[33] G. Weiss, *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*. Boston, MA: MIT Press, 2001.
[34] M. Wooldridge, *Introduction to Multi-Agent Systems*. New York: Wiley, 2002.
[35] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 2nd ed. Upper Saddle River, NJ: Prentice-Hall, 2003.
[36] D. Milojicic, F. Douglis, and R. Wheelter, *Mobility: Processes, Computers, and Agents*. Reading, MA: Addison-Wesley, 1999.
[37] E. J. Chikofsky and J. H. Cross, II, "Reverse engineering and design recovery: A taxonomy," *IEEE Softw.*, vol. 7, no. 1, pp. 13–17, Jan. 1990.
[38] T. M. Austin and G. S. Sohi, "Dynamic dependency analysis of ordinary programs," in *Proc. 19th Annu. Int. Symp. Comput. Archit.*, Brisbane, Qld., Australia: ACM Press, 1992, pp. 342–351.
[39] T. Sandholm and Q. Huai, "Nomad: Mobile agent system for an Internet-based auction house," *IEEE Internet Comput.*, vol. 4, no. 2, pp. 80–86, Mar./Apr. 2000.
[40] V. Roth. (2002). Empowering mobile software agents [Online]. Available: citeseer.ist.psu.edu/roth02empowering.html
[41] T. Eisenbarth, R. Koschke, and D. Simon. (2003). Locating features in source code. [Online]. Available: citeseer.ist.psu.edu/eisenbarth03locating.html
[42] H. Safyallah and K. Sartipi, "Dynamic analysis of software systems using execution pattern mining," in *Proc. 14th IEEE Int. Conf. Program Comprehension (ICPC 2006)*, Washington, DC: IEEE Comput. Soc. Press, pp. 84–88.
[43] N. Suri, "Nomads and agile computing," presented at the US Army CERDEC Intell. Agents Sub Integrated Product Team, Fort Monmouth, NJ, Aug. 2005.
[44] D. Xu, J. Yin, Y. Deng, and J. Ding, "A formal architectural model for logical agent mobility," *IEEE Trans. Softw. Eng.*, vol. 29, no. 1, pp. 31–45, Jan. 2003.
[45] V. R. Lesser, "Evolution of the GPGP/TæMS domain-independent coordination framework," in *Proc. 1st Int. Joint Conf. Auton. Agents Multiagent Syst. (AAMAS 2002)*, M. Gini, T. Ishida, C. Castelfranchi, and W. L. Johnson, Eds. Barcelona, Spain: ACM Press, Jul., pp. 1–2.
[46] T. Wright, "Naming services in multi-agent systems: A design for agent-based white pages," in *Proc. 3rd Int. Joint Conf. Auton. Agents Multiagent Syst.*, 2004, pp. 1478–1479.
[47] K. Sycara, J. Lu, M. Klusch, and S. Widoff, "Dynamic service matchmaking among agents in open information environments," *J. ACM SIGMOD Rec.*, vol. 28, pp. 47–53, 1999.
[48] E. Sultanik, D. Artz, G. Anderson, M. Kam, W. Regli, M. Peysakhov, J. Sevy, N. Belov, N. Morizio, and A. Mroczkowski, "Secure mobile agents on ad hoc wireless networks," in *Proc. 15th Innovative Appl. Artif. Intell. Conf.*, American Association for Artificial Intelligence, Acapulco, MX, Aug. 2003.
[49] Y. Amir, C. Nita-Rotaru, J. Stanton, and G. Tsudik, "Secure spread: An integrated architecture for secure group communication," *IEEE Trans. Dependable Secure Comput.*, vol. 2, no. 3, pp. 248–261, Jul.–Sep. 2005.
[50] T. Clausen and P. Jacquet, "Optimized link state routing protocol," IETF Network Working Group, Tech. Rep. RFC 3626, 2003.

**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, in 1989, and the Ph.D. degree in computer science from the University of Maryland, College Park, in 1995.

He is currently a Professor of computer science at Drexel University, Philadelphia, PA, with joint appointments in the Department of Mechanical Engineering and Electrical and Computer Engineering, where he is also the Director of the Applied Communications and Information Networking Program. He is also a Senior Scientific Adviser to the Department of Justice's Communications Technologies Center of Excellence. His current research interests include several computer science and engineering fields such as artificial intelligence, solid modeling and graphics, computer-aided design (CAD)/computer-aided manufacturing (CAM) integration, mechanical design, and wireless networks. His research has been sponsored by a wide variety of organizations. He has filed four patent and has authored or coauthored more than 150 technical publications.

Prof. Regli was the recipient of many awards, including the National Science Foundation (NSF) CAREER Award, the National Research Council (NRC) Postdoctoral Award, the National Institute of Standards and Technology (NIST) Special Service Award, and the Drexel College of Engineering Research Award. He was the corecipient of the Army's 2006 International Collaboration and the Institute for Defense and Government Advancement (IDGA)'s Best Network-Centric Warfare (NCW) Program Award. He is a Life Member of the Association for the Advancement of Artificial Intelligence (AAAI) and the Sigma Xi. He is a Senior Member of the Association for Computing Machinery (ACM).

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

He is currently an Electronics Engineer/Research Scientist and a Technical Manager with the Command and Control Directorate, U.S. Army Research, Development and Engineering Command (RDECOM), Communications-Electronics Research, Development and Engineering Center (CERDEC), Fort Monmouth, NJ. 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 (AFCEA), the Association of the United States Army (AUSA), and the U.S. Naval Institute (USNI). He is currently the Chair of the Intelligent Agents Integrated Product Sub-Team (sub-IPT), RDECOM Network IPT, the Technical Manager of several Exploratory Development and Advanced Development Programs including the Information Dissemination and Management for Battle Command Services of the U.S. Army Technology Objective Program called Tactical Information Technology for Assured NetOps (TITAN).

**Christopher J. Dugan** received the M.S. degree in computer science from Drexel University, Philadelphia, PA, in 2006.

He is currently with Drexel University, where he was involved in the field of agent system formalizations and distributed voting in the Secure Wireless Agent Testbed Laboratory.

**Joseph B. Kopena** is currently a graduate student of computer science at Drexel University, Philadelphia, PA. He is currently a Researcher in the Secure Wireless Agent Testbed Laboratory, Drexel University, Philadelphia, PA. His current research interests include knowledge representation and wireless networking.

**Robert N. Lass** (S'06) received the Undergraduate degree in 2003 from Drexel University, Philadelphia, PA, where he is currently working toward the Ph.D. degree.

He is a Graduate Research Fellow at the Applied Communication and Information Networking Laboratory, Drexel University. His current research interests include constraint reasoning, distributed systems, and mobile *ad hoc* networks.

**Pragnesh Jay Modi** passed away shortly after the completion of the reference model. During his relatively short career he made several significant contributions to artificial intelligence, especially in the field of distributed constraint optimization. Jay was selected as one of the top ten young artificial intelligence (AI) researchers by the IEEE Intelligent Systems Advisory Board and was also an National Science Foundation (NSF) CAREER Award recipient. He was a much loved professor, mentor, and friend.

**William M. Mongan** (S'06) received the Undergraduate degree in computer science from Drexel University, Philadelphia, PA, 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. His current research interests include software architecture and composition, service-oriented architectures, agent-based systems, and program comprehension through software engineering. He was engaged in computer science education and engineering education. He has taught and volunteered with students in grades 5 through 12. He is an Instrument-Rated Private Pilot of single- and multiengine airplanes.

Dr. Mongan was a Fellow of the National Science Foundation (NSF) 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.

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

He is currently a Senior Lecturer in the Department of Computer Science, Drexel University, Philadelphia, PA. His expertise is in database systems and software design. He is the author or coauthor of many books on a variety of subject matters within and outside of computer science. He was 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. He is currently working toward the Ph.D. degree in the Drexel Applied Communications and Information Networking Program, Drexel University, Philadelphia, PA.

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

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