Agent Systems Reference Model Release Version 1.0a

Project ACIN[1]
DoD Contract #DAAB07-01-9-L504

Document #: 1.0a • Date: 2006-11-20 13:15:19 -0400 (Mon, 20 Nov 2006)
Built from Working Revision 9345

This document has been developed under the Applied Communications and Information Networking (ACIN) Program, DoD Contract #DAAB07-01-9-L504 to the US Army Communications and Electronics Command Research Development and Engineering Center (CERDEC) in support of the Intelligent Agents Sub-IPT.

Waterfront Technology Center
200 Federal Street, Suite 300
Camden, NJ 08103

[1] http://www.acincenter.org

Editors:

Israel Mayk
William C. Regli

Contributing Authors:

Brandon Bloom
Christopher J. Dugan
Tedd Gimber
Bernard Goren
Andrew Hight
Moshe Kam
Joseph B. Kopena
Robert N. Lass
Israel Mayk
Spiros Mancoridis
Pragnesh Jay Modi
William M. Mongan
William C. Regli
Randy Reitmeyer
Jeff K. Salvage
Evan A. Sultanik
Todd Urness

Authority

This draft document is being developed by the Intelligent Agents Integrated Product Sub-Team (IA Sub-IPT) of the Networking Integrated Product Team (Networking IPT) of the US Army Communications and Electronics Command Research Development and Engineering Center (CERDEC). The document represents the consensus technical agreement of the participating IA Sub-IPT Member Agencies, Companies and Institutions. The goals and procedures of the IA Sub-IPT are detailed in its charter document as part of the Network IPT. The record of participation in the development of this document is available from the chair of the Intelligent Agent Sub-IPT at the address below.

Chair
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
Fort Monmouth, New Jersey 07703

Statement of Intent

This document is a technical recommendation for a reference model for those who develop and deploy systems based on agent technology. As such, it

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

The Agent Systems Reference Model (ASRM) allows existing and future agent frameworks to be compared and contrasted, as well as providing a basis for identifying areas requiring standardization within the agents community. As a reference model, the document makes no prescriptive recommendations about how to best implement an agent system, nor is its objective to advocate any particular agent system, framework, architecture or approach.

Through the normal process of evolution, it is expected that expansion, deletion or modification of this document will occur. Current versions of this document are maintained on the Project ACIN (Applied Communication and Information Networking) website:

`https://www.swat.acincenter.org/pubdocs/wiki`

This site is password protected. To request a username and password, or for questions relating to the status of this document, please e-mail:

`agent-ref-model@lists.cs.drexel.edu`

Document Control

Document	Title	Date	Status and Substantive Changes
Initial Document	DRAFT: Intelligent Agent Systems Reference Model	August 31, 2005	Release and review at August 31–September 1 CERDEC IA Sub-IPT meeting.
Quarterly Release	DRAFT: Intelligent Agent Systems Reference Model	October 31, 2005	Release and placement on IASRM Wiki.
Document # 1, Revision 6698	DRAFT: Intelligent Agent Systems Reference Model	December 06, 2005	Second release to the CERDEC IA Sub-IPT
Version 6709:6765M	DRAFT: Intelligent Agent Systems Reference Model	December 19, 2005	Release post December IA Sub-IPT meeting.
Version 7183	DRAFT: Intelligent Agent Systems Reference Model	February 9, 2006	Release for February IASRM meeting.
Version 7574	DRAFT: Agent Systems Reference Model	April 3, 2006	Release for April IA Sub-IPT meeting.
Version 9373:9374M	Agent Systems Reference Model	November 20, 2006	Release Version 1.0a.

Contents

List of Figures

Chapter 1

Introduction

1.1 Purpose and Scope

This document is a technical recommendation for a reference model for those who develop and deploy systems based on agent technology. As such, it

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

The Agent Systems Reference Model (ASRM) allows existing and future agent frameworks to be compared and contrasted, as well as providing a basis for identifying areas that requiring standardization within the agents community. As a reference model, the document makes no prescriptive recommendations about how to best implement an agent system, nor is its objective to advocate any particular agent system, framework, architecture or approach.

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[1]. A reference model drives the implementation of multiple architectures in the same way that a reference architecture drives multiple designs, or a design drives multiple implementations (see Figure 1.1).

1.2 Basis

The basis for this effort follows the approach advocated by the International Standards Organization (ISO)[2] 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[3].

[1] See definition for "architecture" on page 11.

[2] `http://www.iso.org/`

[3] `http://www.feapmo.gov/`

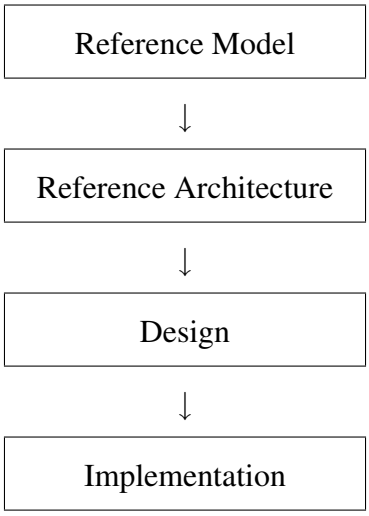


Figure 1.1: Role of a reference model: drives creation of one or more reference architectures, which drive the creation of one or more designs, which, in turn, drive the development of one or more implementations.

This document draws heavily on the reference modeling approaches utilized by several approved international standards:

- ISO 14721:2003, also known as Consultative Committee for Space Data Systems (CCSDS) 650.0-B-1, "Reference Model for an Open Archival Information System (OAIS)". Blue Book. Issue 1. January 2002.
- ISO/IEC 7498-1:1994 "Open Systems Interconnection Basic Reference Model".

1.3 Approach

The approach taken to create the reference model in this document is based on a forensic analysis of existing agent systems. As noted by the definition on page 11, an agent 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," this document offers a different approach from the largely inconclusive debates of the past: the reference model developed in this document is based on static and dynamic software analysis of fielded agent systems. Hence, an agent system describes a software platform for *both* building agents *and* supporting their communications and collaboration within systems. An example of static analysis applied to a portion of the NOMADS agent framework is shown in Figure 1.2.

There are many products in the marketplace today that are marketed as agent frameworks from various sources: from companies, from academia and from 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 document takes a quantitative and evidentiary approach; if it can be built with one

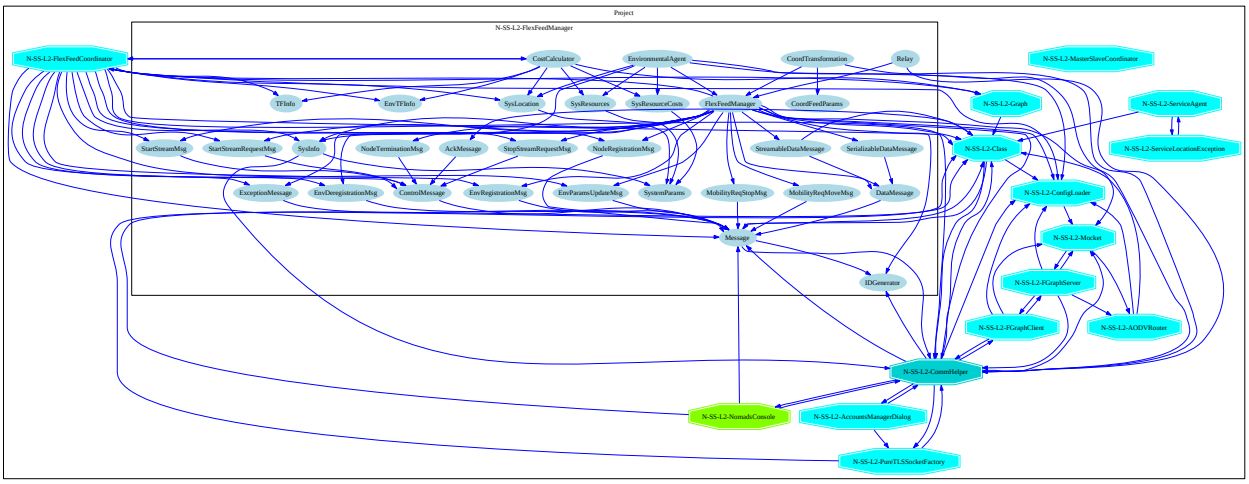


Figure 1.2: An example of software analysis on an existing agent framework. In this case, the figure shows a functional breakdown of the FlexFeed subsystem of the NOMADS agent framework.

of these systems, an artifact might be called an "agent." Anyone building a new agent framework recreates or reproduces some portion of the components in these frameworks (*i.e.*, to enable communication, to enable agent startup and shutdown, *etc.*). Hence, by analyzing existing frameworks and the agent systems they can be used to build, this reference model documents the existing state-of-the-art for what the community believes is an "agent."

The ASRM documents a 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 important to note, however, 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 want or need to have in an agent platform. For example, security for mobile agent code is currently a vastly challenging problem lacking 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. Agents could be implemented in COBOL on an IBM 360 to control smoke detectors just as well as they could be implemented in Java to interact with air traffic controllers using cognitive models of air traffic control behavior. The ASRM makes no assumptions about the internal processing structure of an agent. An agent could be built with a cognitive model (*e.g.*, ACT-R), a neural network that monitors a physical process, just as well as an expert system working to support an automated voice/telephone "help" line. 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.

1.4 Applicability

The ASRM is applicable to existing agent frameworks and will be applicable to all those in the future, as well. The ASRM is designed by extracting commonalities between existing frameworks and by applying existing agent research. The ASRM provides a common ontology for agent frameworks. Previous systems benefit from this model since a language is provided for describing the advantages and disadvantages of each system. Developers of future frameworks will have a blueprint to follow, forming the basis on which all agent systems—past, present, and future—are compared.

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 reverse engineering techniques specifically geared towards agent systems can be applied in a similar fashion to other systems. Further, the techniques themselves can be studied and improved. Agent frameworks are very similar to operating systems, so these reverse engineering techniques also apply in this realm.

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 independently developed software agents and agencies capable of interoperating in a heterogeneous environment.

1.5 Rationale

The motivation for this project is in part analogous to the previous need for a communications reference model, which has since been adopted in many related disciplines. 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 [74]. The primary issue was what this reference model should contain. Unfortunately, many of the existing reference models were not open for inspection. Nevertheless, a 7-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 growth of agent frameworks. Currently, enough openly-available frameworks exist such that the construction of a reference model is possible.

A reference model is behavioral and domain-oriented. A reference model is not a reference architecture. For example, one cannot define conflict resolution in simple Unified Modeling Language (UML). The reference model motivates languages (such as UML, XML) and architectures. In addition, the model includes requirements and supports behavior patterns in addition to the simple structure (e.g., architecture) of the system. The goal of this effort is to abstract the architectural details into concepts.

This reference model facilitates the 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. According to [54], a reference model achieves this payoff by providing appropriate abstractions, simplifying problem solving and providing patterns of the solution for software developers. It is essential and commonplace to create compelling reference models in all fields of knowledge.

1.6 Conformance

A conforming agent system implementation supports the layered model described in Chapter 3 and its agent framework supports the functional model described in Chapter 4. Chapter 6 provides an idealized view of what elements are contained in a conforming reference model. The Agent Systems Reference Model does not define or require any particular method of implementation of these concepts. A conforming ASRM agent framework needs to fulfill certain responsibilities; however, due to the great variety of agent technology and differences in agent system architectures, this document does not attempt to specify any minimal or sufficient set required to be deemed conforming. Where appropriate, examples of the mechanisms that may be used to discharge the functional components and modules identified in Chapter 4 are provided. Use of these, or any other, specific mechanisms are not required for conformance.

It is assumed that implementers will use this reference model as a guide while developing a specific implementation to provide identified services and content. This document does not assume or endorse any specific computing platform, system environment, system design paradigm, system development methodology, database management system, database design paradigm, data definition language, command language, system interface, user interface, technology, or media required for implementation.

A conforming ASRM agent system may provide additional services to users beyond those minimally required of an agent system as defined herein. The Agent System Reference Model is designed as a conceptual framework discussing and comparing agents, agent frameworks, internal and external agent architectures as well as agent-based systems (*i.e.*, systems built with agent technology). As such, it attempts to address all the major needs and activities for agent operating within the context of a multi-agent system in order to define a consistent and useful set of terms and concepts. A standard and other documents that conform to this reference model shall use the terms and concepts defined in the Agent Systems Reference Model in the same manner.

1.7 Related Efforts

An excellent example of an existing reference model is the ISO Open Systems Interconnection (OSI) reference model that 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/Internet Protocol (TCP/IP), Appletalk, or even create their own protocol for each layer, while still remaining true to the OSI model.

In the same way, this reference model is not an attempt to impose (or even define) standards for the implementation of agent systems. It is an attempt to describe the abstract functional layers

and components of such a system.

In the area of agent systems, a reference model for mobile agent systems was constructed in [62]. 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 architecture independent. For example, [62] presents a set of minimum feature requirements (e.g. one of the required components, the *Agent Execution System* supports mobility, communications, agent serialization and security).

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

While there is some resemblance between the FIPA Abstract Architecture and the ASRM. However, a reference model is a further abstraction of an abstract architecture. The Agent Systems Reference Model defines terms, describes concepts and identifies functional elements in agent systems. The ASRM allows 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.

Web Services standards were also examined in the background study of standards. While the W3C Web Services Architecture does not define all the standards compliant frameworks must use, it does prescribe a number of specific standards for interoperability (WSDL, SOAP, etc), whereas the ASRM is standard independent. The W3C Web Services Architecture also describes an architecture that can be used to drive the creation of WS system designs, whereas the ASRM does not make any prescription of the architecture of an agent system. Web services are part of the study, but more specifically, the focus is on agent systems, which are systems that may be used to deliver web services. Not all web services are agents.

The standards that were analyzed fall roughly into five categories. For a more detailed description of any of these standards, refer to Appendix A.

General. Efforts that are higher-level—and therefore less specific—than the ASRM.

- FIPA Abstract Architecture [11]: Defines an agent platform's architectural entities and their relationships, providing standards for interoperability amongst different agents and agent systems;
- FIPA Nomadic Application Support [30]: Describes a monitoring agent and a control agent for restrictive mobile computing environments, as well as an ontology for supporting nomadic computing; and,
- FIPA Agent Management [20]: Describes the module where agent creation, registration, locating, communication, migration, and retirement occurs;

Communications. Efforts dealing with both the actual logistics of communications and the information being communicated. These are mostly protocol-defining standards, and therefore define processes at a much lower level than the ASRM, however, nothing precludes these

standards from working in conjunction with the ASRM. For example, these protocols might be implemented inside an agent framework to fulfill the messaging and mobility functional concepts described in Chapter 4.

- KIF [38]: A standard for encoding knowledge in first order logic;
- KQML [51]: A standard for knowledge exchange;
- FIPA SL Content Language [27]: Describes the syntax for the FIPA Semantic Language content language;
- FIPA Communicative Acts Library Specification [24]: A description of all the FIPA communicative acts;
- FIPA ACL Message Structure [13]: Describes specifications for FIPA ACL message parameters;
- FIPA Message Transport Service [29]: Describes transportation of messages between compliant agents;
- Common Logic (CL) [67] ISO Draft Standard for an information exchange and transmission language;
- SOAP [41] A protocol for exchanging messages encoded in XML, mainly to facilitate interoperability between software objects;
- Hypertext Transfer Protocol (HTTP) [9] A request / response protocol designed for the distribution of "hypermedia," although it is also used for other purposes;
- Web Services Description Language (WSDL) [6] A formal XML-based format for describing the public interface to web services; and,
- Business Process Execution Language for Web Services (BPEL4WS) [2] A formal language for specifying business processes and business interaction protocols, formed by combining WSFL and XLang, languages from IBM and Microsoft respectively.

It is quite likely that one or more of these communication standards will be implemented in a framework architecture. They will help standardize intra-framework communication and facilitate inter-framework communication. Compatibility will be direct result. The ASRM does not specify which standards, if any, are used. Implementation is at the developers discretion and the reference model merely suggests the need for a communication module.

Communication, independent of the standard chosen, can be used for coordination. FIPA extends its domain by standardizing several coordination protocols that can be used in negotiation, resource allocation, or just simple message passing.

Agent Coordination Protocols. Protocols for coordinating the actions of multiple agents; these could be used to fulfill the "agent administration" and "conflict management" functional concepts.

- FIPA Propose Interaction Protocol [31]: describes the process by which an agent suggests to other agents the actions that it wishes to complete;

- FIPA Request Interaction Protocol [33]: defines the protocol that an initiator agent uses to ask another agent to perform some action;
- FIPA Query Interaction Protocol [26]: a protocol that details how one agent requests to perform some action on another agent;
- FIPA Request When Interaction Protocol [34]: similar to the Request Interaction Protocol, this allows an agent to specify a time at which it wishes for another agent to perform some action;
- FIPA Contract Net Interaction Protocol [16]: this protocol defines a process for submitting a job to available contractors, submission and selection of a single proposal, and the completion of the job;
- FIPA Iterated Contract Net Protocol [28]: similar to the Contract Net Interaction Protocol, except contracts can be negotiated prior to selection;
- FIPA Brokering Interaction Protocol [15]: describes the process by which a brokering agent is used by one agent to find resources provided by an other agent and then facilitate the use of those resources;
- FIPA Recruiting Interaction Protocol [32]: similar to Brokering, this protocol allows direct communication between the agent requesting a resource and the agent providing the resource; and,
- FIPA Subscribe Interaction Protocol [35]: allows an agent to request updates concerning resources owned by another agent.

Again, the number of coordination protocols is not specified by the ASRM. Specifying it suggests the need for coordination protocols because coordination is necessary in a MAS. An uncoordinated MAS is perfectly acceptable and still complies with the reference model just as a fully coordinated system that implements all of the above standards.

Once communication protocols are selected and the proper coordination protocols are chosen, it is necessary to specify actual messages at a lower level. FIPA defines a general message and envelope architecture in its representation and encoding standards. These standards operate at a very low-level and describe the format of a message as it is being sent from one agent to another.

Representation and Encoding. Protocols for encoding messages for transmission, or performing protocols over existing communications channels.

- FIPA Bit Efficient ACL [12]: a low-level protocol for general communication;
- FIPA String ACL [18]: a high-level, string representation of the Bit Efficient protocol;
- FIPA XML ACL [19]: a more structured, high-level protocol for message passing in XML;
- FIPA IIOP Transport Protocol (Inter-Orb Protocol) [23]: a message passing protocol involving messages and envelopes all transferred as one single unit of data;
- FIPA HTTP Transport Protocol [14]: a protocol for sending messages via HTTP;

- FIPA XML Message Envelope [22]: the XML encoding for message envelopes that is independent of the type of transport protocol used;
- FIPA Bit Efficient Envelope [21]: the low-level bit encoding for message envelopes that is independent of the type of transport protocol used;
- Extensible Markup Language (XML) [4] A meta-language for describing other markup languages, intended to facilitate sharing of data across heterogeneous systems;
- Web Ontology Language (OWL) [56], a markup language to facilitate ontology sharing on the World Wide Web;
- Resource Description Framework (RDF) [44] A general purpose language for representing information on the web; and,
- RDF in XML [3].

The representation and encoding standards specify how messages should be formatted and possible encodings that can be used for transmission. These processes are very low-level and far beyond the scope of the reference model. No communication protocol is specified so its representation is certainly omitted as well.

Lastly, FIPA and others provide a vocabulary for the messages. These ontologies restrict the contents of a given message to a finite ontology that should be used by most agents in a framework.

Ontologies. Standards for knowledge representation. These might be utilized in the "agent reasoner" component of the agent model presented in Chapter 3.

- FIPA Device Ontology [25]: used to specify resource information (such as size or availability) in inter-agent communication;
- FIPA QoS Ontology [17]: provides a vocabulary for communication concerning the quality of service that can be provided by agent services and resources;
- Dublin Core Metadata Initiative [1]: provides abstract online metadata standards that are business model independent;
- IEEE Standard Upper Ontology [47]: describes an ontology that can be used for data interoperability, information search and retrieval, automated inferencing, and natural language processing.

Once again, no ontology is specified in the ASRM; however, the need for an ontology is noted and even suggested. The overall goal of the ASRM is to specify areas where standards may be necessary and should be implemented, but it makes no prescription as to which ones should be used.

Chapter 2

Terminology

This section puts forth a set of formalisms, terminology and definitions to be used for the remainder of this document. Where appropriate, citations to relevant sources in the literature are given. It is not the goal of this section, however, to be completely compatible with existing literature. This is impossible as existing literature from the field does not offer complete and consistent definitions. In many cases there are considerable conflicts and overloaded terms. Where possible, this document attempts to be consistent with the most generally accepted conventions of use of terms.

2.1 Definitions and Acronyms

Note that terms are shown in bold and defined when first used in the text. An index is also provided at the end of this document.

Action: An event triggered by an agent that may affect its environment (see **agent environment**).

Adversary: An agent whose goal conflicts with another agent's goal.

Agent: An agent is a situated computational process with one or more of the following properties: autonomy, proactivity and interactivity. This topic is discussed at length in Chapter 3.

Agent Architecture: Modules and structure for designing and implementing individual agents. This reference model abstracts these internals into three components: a **sensor interface**, a **effector interface** and a **reasoner**. The reasoner inside an agent architecture might include a knowledge base, workflow monitor, or planner.

Agent Complexity: The level of interaction sophisticatation and computational complexity of the internal agent architecture.

Agent Environment: See Enivornment.

Agent Framework: A software component that supports the execution of agents. In some agent systems, the agent framework may be trivial, if the agents run natively on the **platform** (as opposed to in a virtual machine or some other local execution environment). However, most agent systems are based on a framework that supports key functionality commonly needed by all agents, such as services for migration, agent messaging, and matchmaking.

Agent Framework Architecture: The structure for the design of an agent framework, including implementation specifics of how the required and optional functional components of the agent framework are to be implemented and inter-connected.

Agent Group: A set of one or more agents.

Agent Infrastructure/Infrastructure: The combination of the **host** and platform on which the agent framework and agents execute.

Agent Messaging: The means by which agents communicate. Agent messaging is sometimes implemented by deploying a class of mobile agents to act as a message delivery system. In some domains and **agent system architectures**, reliable message delivery is not ensured (or even feasible). The OSI reference model provides a general framework for discussing these communications.

Agent Organization: An arrangement of communications and interaction constraints among agents in an agent group. Examples of agent organizations include peer-to-peer and hierarchical structures.

Agent System: A system in which the functionality is implemented by one or more **agents**.

Agent System Architecture: The structure for the design of an entire agent system, including how agents interact, configuration of infrastructure and selection of frameworks.

Agent Team: An agent group in which all of the agents have common goals.

Agent-Based System: A system that includes agents as a substantial aspect of its functionality. Note an agent system is necessarily agent-based, but an agent-based system may include other systems not based on agents.

Architecture: A set of abstract patterns that dictate the design of a software system.

Autonomy/Autonomous: The characteristic of being self-sufficient, independent, or self-controlling.

Belief: An agent's position as to the state of a proposition, usually influenced by its **Perceptions**.

BNF: The "Backus Naur Form" of a grammar description, based on terminal and nonterminal definitions.

Class: Throughout the majority of this document, this term refers to a group of system components (usually agents) that can be grouped together as possessing common characteristics. In places this term refers to the accepted Software Engineering definition: an object-oriented **software component** consisting of a collection of methods and variables that may be instantiated as an object.

Communication: The transmission of information between entities (*e.g.* agents). In the case of stigmergic systems, such transmission might occur through an intermediary medium, such as the environment.

Complexity: Complexity refers to the cost, in time or space, of a system.

Composition: The act or result of combining similar objects to increase complexity (and thereby expressivity). Examples of such objects include data types within a language, plans, and ontologies.

Concept: A software entity comprised of one or more components that performs a high-level business logic function within a software system.

Conformance: See Section 1.6.

Consistent: The state of a set of propositions containing no contradictions.

Continuous: A continuous function or activity is one that can be divided into arbitrarily small units or components. For example, time is a continuous parameter in the environment of an agent-based system.

Control: Deals with the behavior of dynamical systems, built of agents, over time.

Controller/Agent Controller: The software that handles the internal processing for an individual agent, mapping its sensory inputs into output by its effectors.

Cooperative: The state of being helpful not only to one's self, but also others. Note that cooperation *does not* imply that the collective behavior is globally optimal.

Decide: To decide is to render a yes-no answer to a specific query or input (e.g. a theoretical computer science decision problem such as boolean satisfiability (SAT)).

Desire: A state or property of the world that an agent may want to be or be true.

Deterministic: A process that contains no random chance. The outcome of a deterministic process is always predictable, given a description of the process and the state of the world.

Dynamic Analysis: Methods for analyzing software code by processing run-time statistics.

Effector Interface: The interface or API through which an agent acts on the outside world, including the transmission of messages to other agents.

Encryption: A security mechanism that encodes messages in order to make them prohibitively difficult to interpret by unauthorized parties. This process often involves a function based on a shared secret or certificate.

Environment: The world in which an agent is situated from the point of view of the agent. The agent senses and affects its environment through its actions. The environment may be virtual, such as a logical view of the World Wide Web or an virtual market for a combinatoral auction. The environment may also be an abstraction for the physical world, such as a computer network or robotic domain.

Framework: See Agent Framework.

Functional Decomposition: A model of a system or **architecture** in which each "block" (*i.e.* **component**) is described without "and" or "or" qualifiers.

Goal: A **desire** an agent intends to fulfill.

Heterogeneous: Of different types or kinds. A **group** of agents is heterogeneous if it includes agents from at least two different agent frameworks.

Hierarchy: An organizational structure, often used for ranking, that utilizes an *n*-tier schema, with each tier serving as the "parent" to the tier(s) below.

Homogeneous: Of the same types or kind. A **group** of agents is homogeneous if they are all based on the same agent framework.

Host: A physical computing device on which the platform resides and the agent framework exists and executes.

Inference: A proposition drawn from (or implied by) another that is admitted or supposed to be true.

Intelligent Agent: This document makes little distinction between agents, as a general concept, and intelligent agents. Hence, an intelligent agent is simply an agent for purposes of this document.

Intent: A commitment toward an **Action** or sequence of actions.

Interact: The act of invoking behavior in or sharing data with another software entity.

Interactivity: Mutually or reciprocally active; of, relating to, or being a two-way electronic communication system.

Itinerary: An ordered set of **Hosts** to which a **Mobile Agent** attempts to **Migrate**.

Knowledge Base: A set of representations of facts about the world.

Layer: An abstraction that is used to partition a system into more independent and cohesive components.

Legacy Software: Software systems at the end of their development life cycle. They are often expensive to maintain but even more expensive or risky to re-write.

MANET: See **Mobile Ad-hoc Network**.

MAS: See **Multi-Agent System**.

Matchmaker: An agent or service capable of conducting **matchmaking**.

Matchmaking: The act of pairing agents and/or services with each other. This is usually motivated by fulfilling a needed capability. Given a description of such a capability, a **matchmaker** must pair the requester with an agent or service that provides the capability.

Mediator: A facilitator of agent interoperability that helps to identify conflicts in agent behavior or language and provides translation or conflict resolution services to the conflicting agents.

Message: An encapsulation of information containing enough meta-data to be transmitted from its source to one or more ultimate destinations, according to some level of **Quality of Service**. For example, a message between agents communicating over the Internet might contain the IP addresses of both the sender and the recipient. On the other hand, agents in a stigmergic system might use pheromones as messages, in which case the ultimate recipient of the messages might neither be known nor required at the time of transmission.

Migration: The process of pausing agent execution, serializing the state of the agent, traveling over the network and resuming execution on a new host.

Mobile Ad-hoc Network: A network in which each host acts as a router. Hosts may be capable of moving, allowing the possibility of a dynamic network topology. Since each host can communicate with its immediate neighbors, routing algorithms and mechanisms are required for global connectivity.

Mobile Agent: An agent capable of **migrating** from one host to another.

Module: A **software component** containing a set of subprograms and data structures/**classes** in a group.

Multi-Agent System: An **Agent System** composed of multiple agents, possibly distributed, and possibly implemented on different architectures.

Network: A physical, hardware, and software medium through which agents are capable of **messaging** and **migrating**. In most networks, there is no guarantee of communication between all pairs of agents. In such instances, messages must be routed between agents for complete connectivity.

Observable: A form of decision process in which the agent has complete information regarding its state and environment when making its decision.

Operating System: The software system responsible for providing platform-level services to the agents.

Operational Abstraction: A formal mathematical description of the observed behavior of a software system.

Package: A **software component** in which related **classes** and interfaces are grouped.

Packet: The **network**'s encapsulation of information. There is not necessarily a one-to-one mapping between packets and the information (*i.e.* **messages**) they transmit; a message may be split into multiple packets for delivery. Likewise, a single packet might contain parts of multiple messages.

Partially Observable: A form of decision process in which the agent does not have complete information regarding its state and environment when making its decision.

Partial Order: The state in which a list of elements are not necessarily ordered with respect to each other. This term is usually used in reference to the ordering of steps/tasks in a **Plan**.

Perception: An atomic unit of information gleaned from the **Environment**.

Permissions Manager: The host or mechanism responsible for trust management in the system. The permissions manager ensures that an agent or host is authorized to perform its requested actions at any given time.

Physical World: The complete world in which the agent is situated, including the host, platform, framework, geography, etc. For example, for a agent embodied in a robot that has to navigate a room, the room would be part of the world in which the agent is situated.

Plan: A course of action, usually devised to achieve a goal. Usually, a plan consists of a set of states that need not be totally ordered.

See **Partial Order**. Each state consists of a set of propositions, including preconditions and postconditions. Plans can be defined for both individual agents and collectives.

Planner: A program, or agent, that is able to generate a **Plan**.

Platform: The collection of all of the software resources atop of which the **Agent Framework** and agents run. Note that software other than the agents and agent framework may run on the platform.

Proactivity (or Proactive): Acting in anticipation of future problems, needs, or changes.

Quality of Service: A guaranteed level of performance in the transfer of information [52]. Abstractly, QoS is considered a contract between two nodes in a network. The contract can even span multiple layers of the OSI reference model [74]; for example, the transport layer might guarantee a lower bound on available bandwidth to a streaming video program running on the application layer.

Reverse Engineering: The analysis of software systems by extracting artifacts and functionality from an existing system.

Routing: Abstractly, the process of choosing the next entity to which to forward an object. This is usually used in terms of networking, in which case it is the process of chosing the next network **Host** in the path of a **Packet** to its destination. In the case of **Mobile Agents**, this is means by which an **Itinerary** is created.

Security: Policies and procedures that enforce authentication, authorization, trust management, and access limits among the agents.

Self-Interested: A property of agent interaction in which an agent selects actions based upon its own utility, as opposed to the utility provided to other agents.

Semantics (or Semantic): The meanings of that which is represented by a formal abstraction (*i.e.* according to a **syntax**).

Semantic Web Service: A service that uses Semantic Web standards to describe its capabilities and for others to access its outputs.

Sensor Interface: The interface or API through which an agent receives input from the outside world, including the receiving of messages to other agents.

Service: A continuous computing process acting largely in response to requests from agents or other services. Examples include a database, a form on the World Wide Web, or a sensor interface.

Situated: Instantiated within an environment or physical world which may be sensed and acted upon.

Socket: The combination of an IP address, a protocol, and a port number [59, 60].

Software Agent: A type of **agent** without a physical embodiment; a program. This in contrast to a human user or robot, which are also agents. Nonetheless, a robot's control software may be constructed out of a set of software agents.

Software Component: The software equivalent of "Hardware Component:" a loosely-defined term that refers to any piece of software that encapsulates functionality. Examples include objects (*e.g.* classes), architectures, frameworks, and design patterns.

Standard: A basis for comparison; a reference point against which implementations are evaluated and validated.

Static Analysis: Methods for analyzing software strictly from the source code.

Stochastic: A process whose outcome is not predictable with absolute certainty.

Stream: A sequence of communicated bits, often over the network, on a file system, or looped on a local host.

Syntax: Formalisms (*e.g.* rules and relations) that govern the valid structure of sentences within a language.

Task: Relating to an event or an activity needed as a pre-requisite to achieve one or more goals.

Tier: This partition of an agent software system defines a physical or logical layer of abstraction within the system. Tier is used in the same context as it is in the ISO model.

UML: The Unified Modeling Language is an open object modeling and specification language. Software engineers may use UML to specify software **architectures**.

View: An architecture description of a software system in a particular context that is relevant to a group of stakeholders, including developers, business-persons, customers, etc.

Web Service: A service that uses W3C standards for the WWW (HTTP, UDDI, etc).

Workflow: A series of tasks that serve as sub-goals toward an agent's objective. Workflows are defined by either an individual **agent** or by a management and control agent.

Wrapper: Software that provides a layer of abstraction above or interface to another piece of software. **Agents** are often used as wrappers to add intelligence and control to existing applications.

2.2 UML Graphical Notation

The Unified Modeling Language (UML) is used to generate most illustrations in this document. UML is an Object Management Group (OMG) standard for modeling software architecture, behavior, and processes. UML is used to create diagrams that show levels of abstraction appropriate to a reference model in a formal, well-defined way, while maintaining widely-accepted readability of the diagram.

A subset of the Unified Modeling Language is chosen to formally describe behavioral and the structural components of agent systems. Each diagram contains entities and relationships appropriate to those entities. These relationships are intuitive; for example, inheritance, containment, and "uses" relationships exist among components in many UML descriptions. These diagrams can contain numerous stencils and illustrations; a brief description of the most important stencils is presented in this section.

Entities and Relationships. Each of the diagrams presents a set of relevant UML entities. Depending on the diagram, these entities may represent artifacts including actions, actors, processes, products, etc. The meaning of the entities is described within the context of the diagram. For example, use cases present certain entities, while component diagrams present other entities; although the two diagrams may describe the same system, they do so from different perspectives because of the entities appropriate to the diagram.

In each diagram, there also may exist relationships between the entities. These relationships are depicted as lines and arrows between the entities. The meaning of these relationships depends on the context of the diagram. For example, a process diagram's relationships indicates data flow or temporal ordering of events, whereas a component diagram's relationships indicates coupling via data sharing, a method call, or data flow. In situations where further description is appropriate to add meaning to the relationships, they are indicated via a ≪`caption`≫ within brackets as shown.

2.2.1 Use Case

The use case UML subset contains actors, use cases, and relationships. Use cases are described by the oval figures in the diagram, and the relationships may vary from use case to use case. Use cases are carried out by one or more actors, described by the stick figure stencil. Relationships include general relations (solid lines), extends, and includes relations. These relations are indicated with dashed lines and a qualifier in angle brackets. See Figure 2.1.

In this discussion, use cases represent the **Scenarios** view of agent software systems.

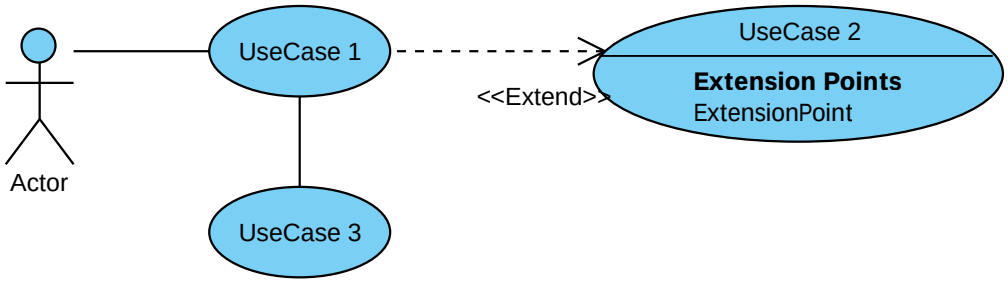


Figure 2.1: Use Case Diagram Legend.

2.2.2 Activity Diagram

Activity diagrams, among others, use Activities to represent high level business-model level processes that take place in the system. These Activities contain Actions that represent the smaller, more focused behaviors that take place to achieve that business process. The result is a flow chart that forks and joins (represented by the black bars) arbitrarily. Start and end points on the flow chart are represented by a solid circle and concentric circles, respectively. See Figure 2.2.

To partition execution modeled by an Activity diagram, **swimlanes** are used. Swimlanes show various Activities executing in parallel or in sequence by multiple actors. It is a way of separating the description into its associated processes. Anything that is appropriate for the diagram may be contained within a swimlane. In Figure 2.2, these swimlanes are depicted via partitions.

In this discussion, Activity Diagrams represent the **Logical** view of agent software systems.

2.2.3 Sequence Diagram

Sequence diagrams show a timeline of events. Lifelines represent particular processes in the system, with the blue vertical bars below them indicating relative times during which process is actively executing. Communication is achieved through message passing (the arrows between the Lifelines), and these messages can represent recursive or returning execution via messages passed within a lifeline, as shown in Message 1 on LifeLine2. See Figure 2.3.

In this discussion, Sequence Diagrams represent the **Process** view of agent software systems.

2.2.4 Component Diagram

Component diagrams are typically used for static UML descriptions, and represent a higher level object diagram. The entities in this diagram are components instead of objects, and they could represent a collection of objects or an entire subsystem. Components, like objects, have inputs and outputs (called ports) that are the basis for relationships between components. Relationships include generalization (the solid arrow with a triangle), composition (the solid line with a concentric cross and circle), and a generic relation (the dashed arrow) that is usually qualified with a

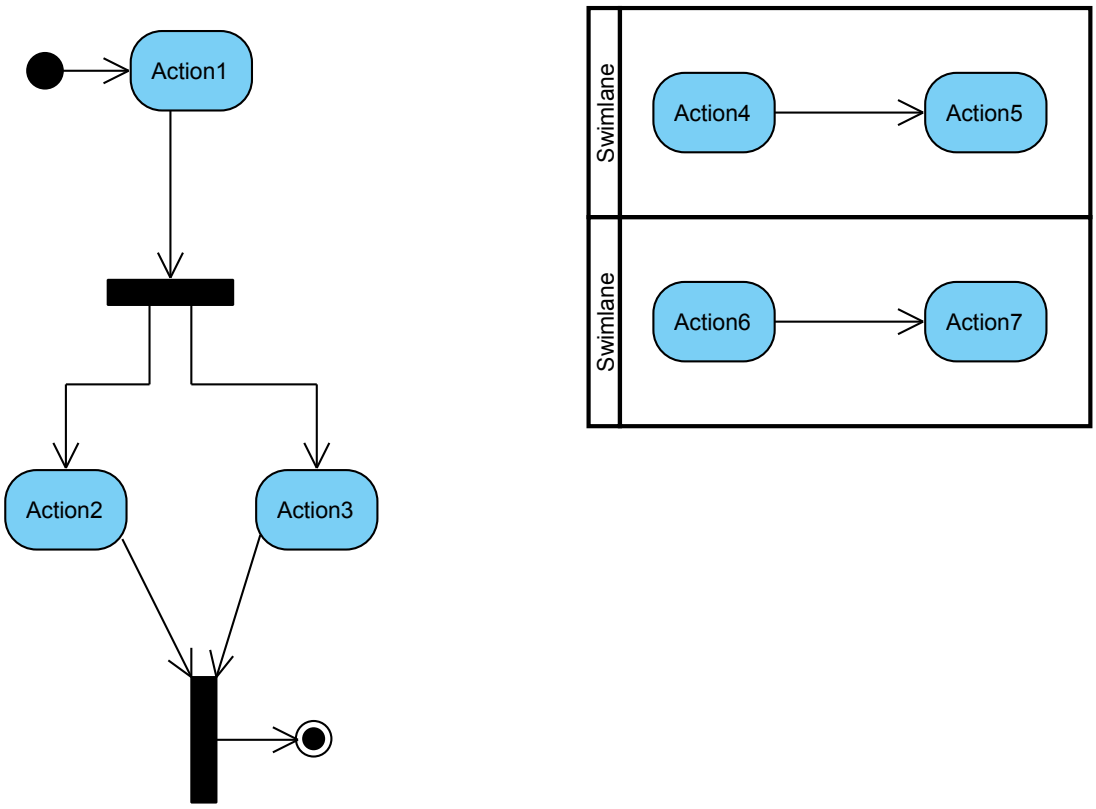


Figure 2.2: Activity Diagram Legend.

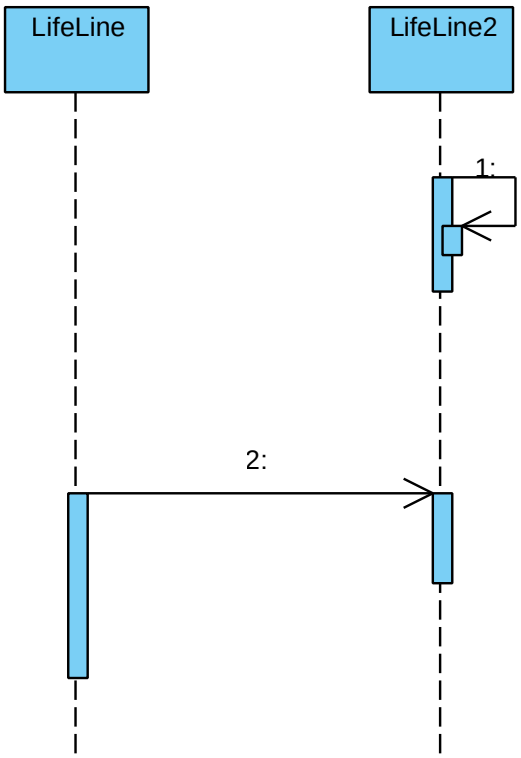


Figure 2.3: Sequence Diagram Legend.

description in angle brackets. Finally, Notes are represented in all diagrams by pieces of paper. See Figure 2.4.

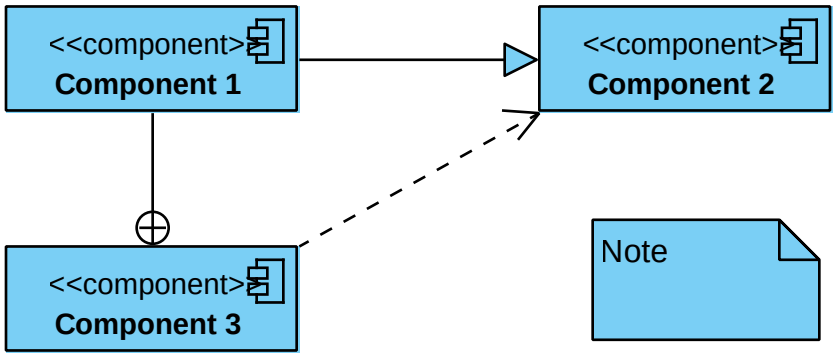


Figure 2.4: Component Diagram Legend.

In this discussion, Component Diagrams represent the **Development** view of agent software systems. It should be noted that in typical **4+1 Model** documentation (see Section 5.2), the Development View represents a software system down to the object level of abstraction. This is certainly possible using UML diagrams, but is too detailed for a reference model. Therefore, component diagrams are used to illustrate development-level concepts in an agent software system. These components are meant as an idealized example of a possible agent software system conforming to the ASRM (described further in Section 6.1).

2.2.5 Component

A Component is a class-level entity containing specific functionality to the system. For example, Quality of Service metrics, Cryptography, and User Authentication are examples of system components. They are the most concrete system components.

2.2.6 Example Implementation

An Example Implementation is an attribute of a Component. It provides more detail about a component by showing domain specific possibilities for the component being described.

For example, Encryption could be considered a Component, but there are many ways to implement Encryption in a system. In fact, some implementations are more appropriate to a particular domain than others. For instance, a public-key scheme is often desirable in multi user systems, whereas a fast point-to-point solution might prefer a shared secret scheme. In either case, the Encryption component is implemented, but there are a variety of ways to carry out that implementation.

In fact, this concept is central to any reference model. The most concrete components still leave a great deal of choice to the developers regarding implementation. Implementation details are intentionally excluded to allow for domain specificity. Instead, slightly to highly abstract views that represent most agent systems in general are presented.

2.2.7 Subsystem

A Subsystem is a collection of components. These components interoperate closely to achieve common or similar functionality. For example, a graph display subsystem contains a number of components (or even other subsystems) to handle parsing the graphic, laying out the graph on screen and physically drawing the pixels. These classes have much higher coupling with one another than with other components in the system; therefore they comprise a subsystem.

2.2.8 Layer

A Layer is the most abstract system component. Layers are package-level components consisting of Subsystems, forming a hierarchical view of the system. Layers are logically stacked on top of one another, and interaction is implied in a top-down manner between adjacent layers of the system. In other words, components existing in one layer may call or use components in the layer directly below it, and return information to components in the layer directly above it.

For example, most operating system designs exhibit a layered structure. In the top layer, user-interaction functionality exists and the user makes requests at a high level (*i.e.*, open a file on the screen). The top layer components send that request to the security layer below to check file permissions, etc. The security layer calls components in the hardware layer to open the file from disk and draw it on screen using the video card. The result is then passed back through the layers to indicate to the user that the command completed successfully. In the event of an error that is initiated at the lower layer (*i.e.*, disk read error), this error is passed back to the top layer for presentation to the user.

2.2.9 Miscellaneous Diagrams

Other UML diagrams may be used to add clarity to the text or to the standard UML subset chosen in this section. These diagrams show actors and other entities as slight variations to show that these diagrams are generic and do not indicate data flow or behavior defined within the ASRM. As an example, the Generic Business Actor (see Figure 2.5) is essentially the same as an actor, but it is depicted differently to show that this is a generic role that is not specifically assumed by agents or agent systems.

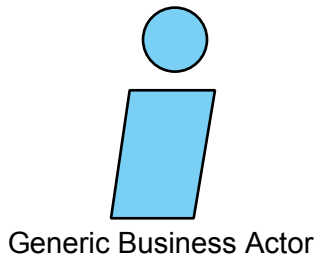


Figure 2.5: Basic Diagram Legend.

2.2.10 Agent UML

Agent UML [45] (AUML) defines a new notation that uses UML as a guide for defining multi-agent systems. This notation includes formal descriptions for agents, roles and group management. The UML subset chosen for this document relies more heavily on standard UML notation, because this document does not assume an understanding of agent groups and roles. Therefore, standard UML notation provides a more primitive description of the concepts and components described. Once these definitions are understood, they can be mapped to standards such as AUML.

Chapter 3

Agent System Concepts and Layers

This chapter 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. Section 3.1 provides a definition of agents and their relationship to agent systems and their interaction with the environment via this model. The layers of structure present in agent systems, mediating between agents and their environment, are then described in Section 3.2. Section 3.3 relates this structure to the OSI model for networking. Broad classifications of agents and agent systems in terms of complexity and level of abstraction are defined in Section 3.4.

Several diagrams are used throughout this chapter and the remainder of the diagram to define and explain these models. Figure 3.1 provides a legend to the symbols used in these figures.

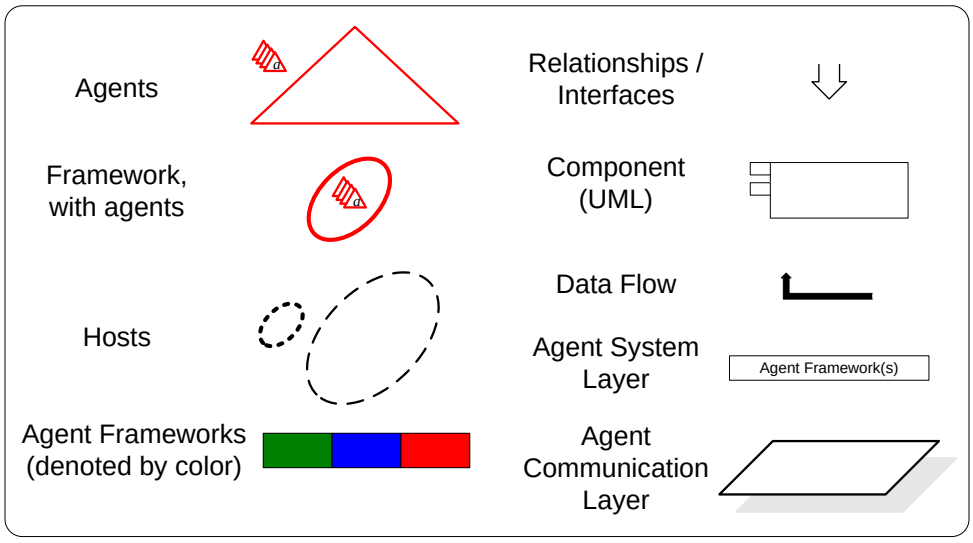


Figure 3.1: Legend for agent system figures and layer diagrams.

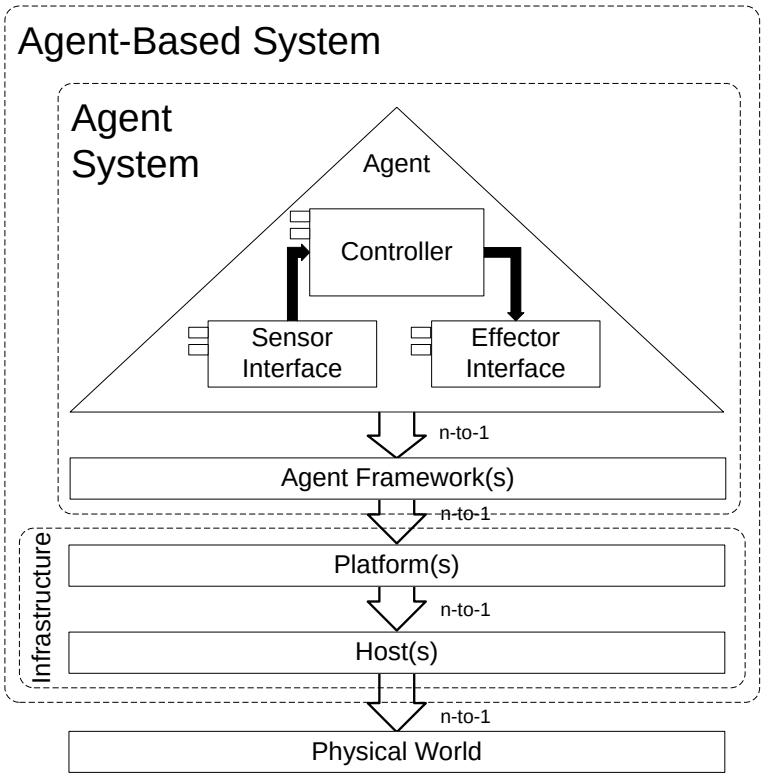


Figure 3.2: Abstract model of an agent system. Such systems decompose into several layers of hardware and software that provide an operating context for agents, situated computational process that sense and affect their environment. Note that the relationships across layers may be *n*-to-1.

3.1 What is Meant by Agent (*i.e.*, What is an Agent?)

Software agents, sometimes called intelligent agents or simply "agents," are **situated** computational processes—instantiated programs existing within an environment that they sense and effect. Figure 3.2 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 the environment. Importantly, the model does not commit sensor and effector interfaces to specific hardware or software structure and form, but rather generically as dataflow 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 document 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 robot controller. In addition to being situated in an environment, one or more of the following properties hold for any agent:

- **Autonomous**. Agents may perform their own decision-making, and need not necessarily comply with commands and requests from other entities.

- **Proactive**. Agents need not wait for commands or requests and may initiate actions of their own accord.
- **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 of 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**, computational processes that exist to provide functionality for use by other processes, do not necessarily exhibit these properties. While by definition interactive, 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:

- **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.
- **Social**. Many agents interact significantly with other agents in achieving their tasks, 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 Chapter 4.
- **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 there continuing execution. Mobility is further classified and described in Chapter 4.

Section 3.4 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.

3.2 Infrastructure for Building and Supporting Agents

As computational processes, agents do not exist on their own but rather within computing software and hardware providing 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 requires significant infrastructure to provide core functionality 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 supporting them. It is comprised of several layers as shown in Figure 3.2 and described as follows:

- **Agents** implement the application, they achieve the intended functionality of the system.

- **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 one extreme, the a framework could even be 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.

- **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.

- **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.

- **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. A **multi-agent system** is an agent-based system that includes 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 is shown in Figure 3.3 of an agent-based system that extends over several hosts. Figure 3.4 shows another example, of devices in the agent system connected at the host layer via wireless networking, transmitting and receiving signals in the environment of the physical world. With respect to the ASRM, communications are abstracted at the platform layer by 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, providing for both execution and access to the world. Figure 3.5 provides several examples of current technology mapped onto these layers.

In general, this document 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.

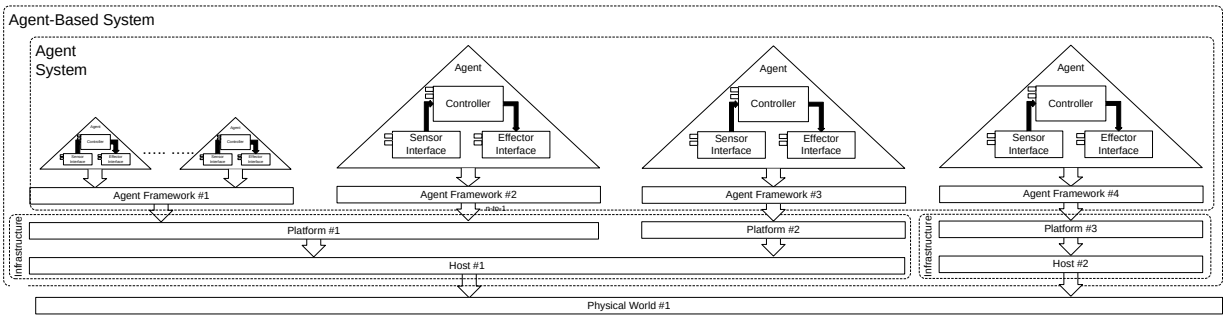


Figure 3.3: Agents and agent frameworks are often part of larger systems. Such larger systems are called agent-based systems and encompass all of the different agents, frameworks, platforms (along with their non-agent software) and hosts needed to deliver the functionality of the system.

3.3 Communication Among Agents

Communication among agents is a critical aspect of many agent systems. As such, the existing Open Systems Interconnection (OSI ISO 7498:1984) reference model is applicable to describing communication among and between agents. As agents are situated within a system, they make use of its communication components. There are several distinct ways in which an agent framework's communication components can be mapped to the OSI reference model.

Figure 3.6(a) shows the established OSI 7 layer communications model. In this model,[1]

- The Physical layer (Layer 1) defines all the electrical and physical specifications for devices and their major functions, including tasks such as establishment and termination of a connection to a communications medium; contention resolution and flow control; and modulation between the representation of digital data in user equipment and the corresponding signals transmitted over a communications channel.

- 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. Any addressing scheme at this layer is physical, which means that the addresses (e.g., MAC address) are hard-coded into the network hardware and the addressing scheme is flat. This is the layer where bridges and switches operate.

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

- 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

[1] Information abstracted directly from the ISO OSI Standard and Wikipedia.

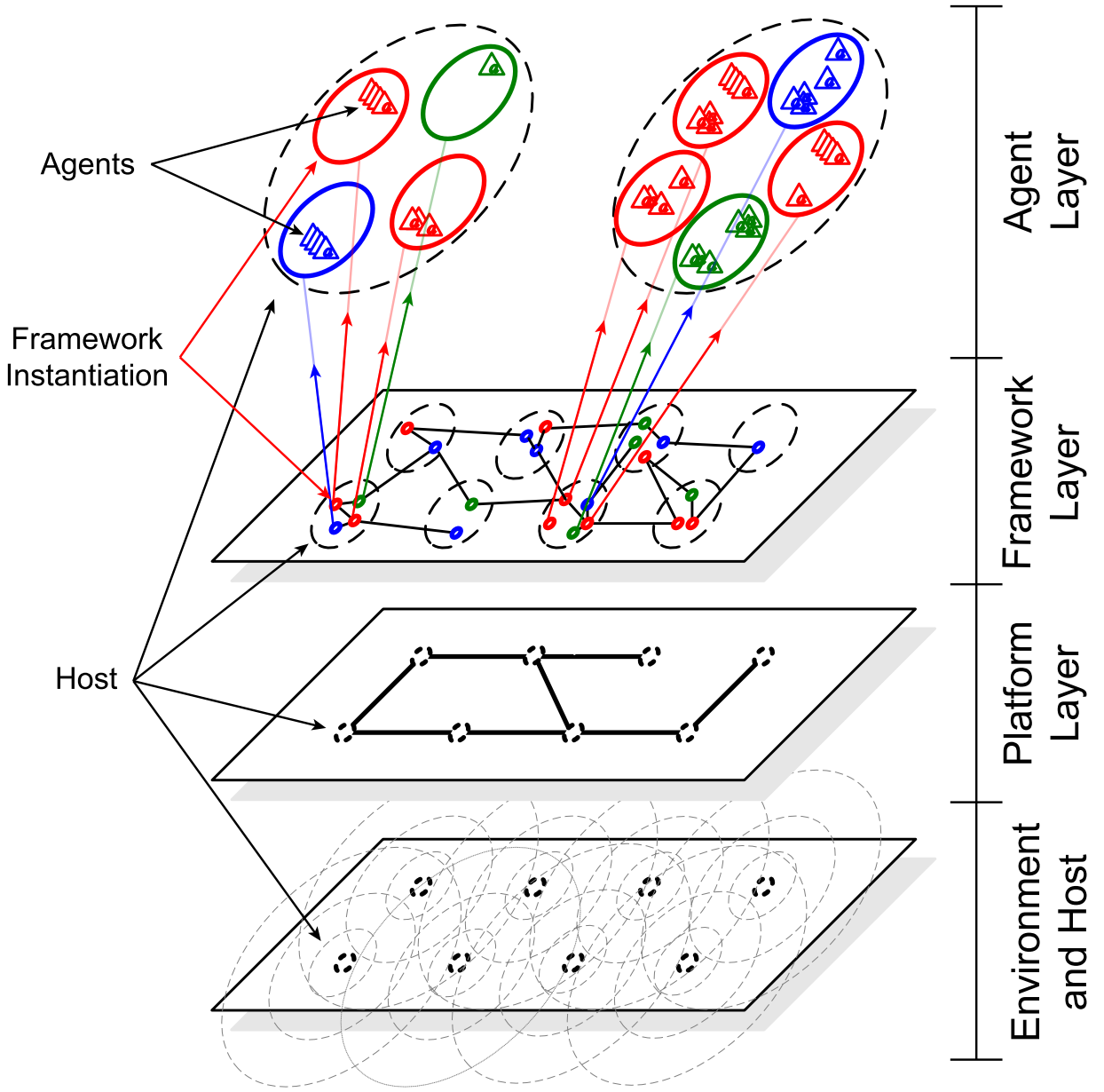


Figure 3.4: Agents are depicted as computational processes running within frameworks supported by platforms and executing on hosts operating together on some network.

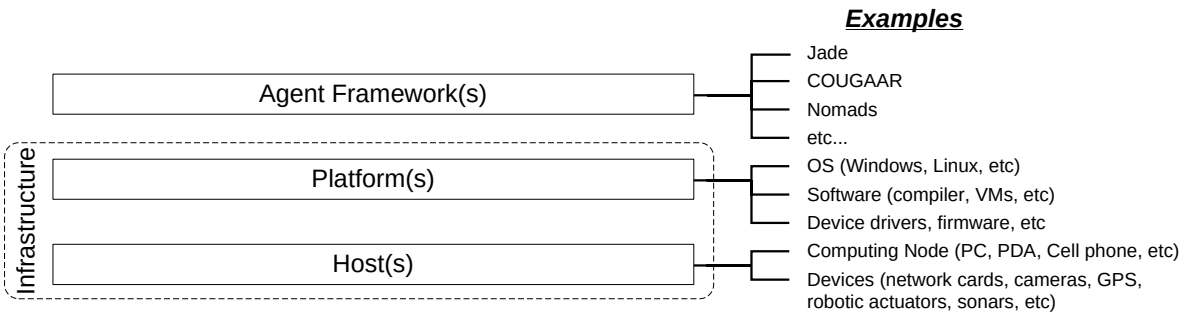


Figure 3.5: Examples of current technologies (circa 2006) mapped onto different layers of agent systems as presented in Figure 3.2.

transfer. The transport layer controls the reliability of a given link. Some protocols are state and connection-oriented, implying that the transport layer keeps track of the packets and retransmits those that fail. The best known example of a layer 4 protocol is TCP.

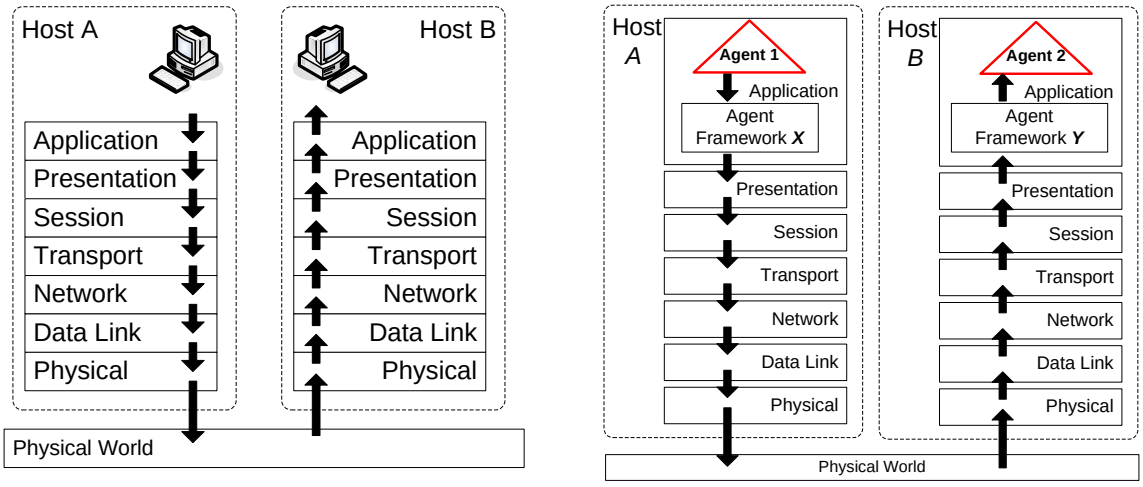
- The Session layer (Layer 5) provides the mechanism for managing the dialogue between end-user application processes. It provides for either duplex or half-duplex operation and establishes check-pointing, adjournment, termination, and restart procedures.

- The Presentation layer (Layer 6) relieves the Application layer of concern regarding syntactical differences in data representation within the end-user systems. MIME encoding, data compression, encryption, and similar manipulation of the presentation of data is implemented at this layer. Examples: converting an EBCDIC-coded text file to an ASCII-coded file, or serializing objects and other data structures into and out of XML.

- 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. Through application layer protocols, software applications negotiate their formatting, procedural, security, synchronization, and other requirements with the network. Some common Application layer protocols are HTTP, SMTP, FTP and Telnet.

Figure 3.6(b) shows one way (perhaps the most common in practice) that the OSI layered model is related to the agent systems reference model. 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 the OSI stack and agents are largely insulated from needing to process information at these layers. However, 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 effect 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



(a) 7 Layer OSI Model.

(b) Agent systems within the OSI model.

Figure 3.6: 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 this layer.

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., Figure 3.2), 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 vacuum cleaner world described in exercise 2.5 on page 51 of [61]) communicating stigmergically through their physical environment.

Other relevant mappings between the ASRM and OSI model include:

- 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.
- Serialization, or other encoding needed for agent mobility, may be considered a Presentation layer functionality and be provided by the agent framework.
- Individual agents could encode or encrypt their own messages, hence assuming the functionality of the Presentation layer.
- 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.

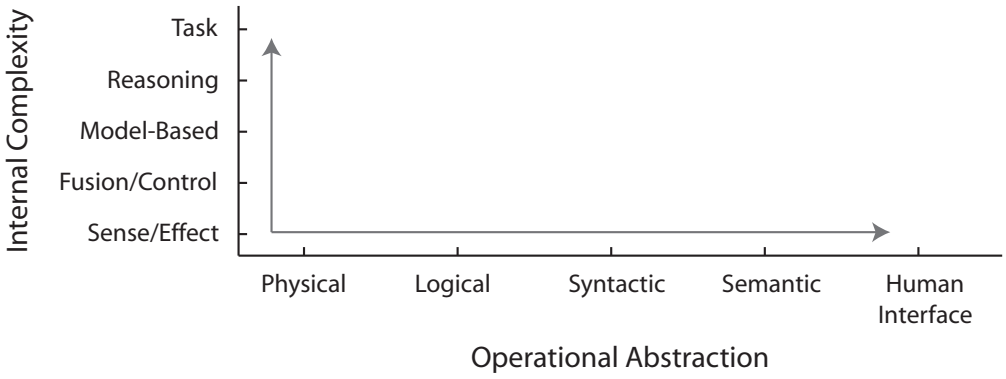


Figure 3.7: Matrix of agent internal complexity and level of operational abstraction.

3.4 Classifying Agents

The complexity of an agent, and subsequently of a multi-agent system built of many agents, may be viewed along at least two different aspects:

- **Internal agent complexity**. The sophistication of the agent's internal reasoning. Agents may be constructed from simple condition-action rules to elaborate deliberative models.

- **Operational abstraction**. 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.

These two aspects outline a classification of agents, as shown in Figure 3.7. 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 also provide a complexity classification of a multi-agent system through an aggregation of the agents in its community.

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

3.4.1 Internal Agent Complexity

The internal complexity axis of Figure 3.7 is a leveling of increasingly sophisticated agent reasoning mechanisms:

- **Sense/Effect** agents are the simplest class of agents, reacting directly to basic environmental stimuli. Examples include agents monitoring smoke detectors or executing stock market limit orders.

- **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.

- **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 inventory levels and issuing resupply requests.

- **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 or determining actions evolving the current state under the agent's world model to match those goals. Agents involved in multi-step service interactions, such as searching and purchasing from online merchants, may have to perform such reasoning when determining what products to consider purchasing and under what conditions to actually purchase those products, authenticate themselves, provide payment information, etc.

- **Task** agents, in addition to having a notion of their own goals, model and reason using 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.

3.4.2 Operational Abstraction

The operational abstraction axis of Figure 3.7 captures the layers of application domains with respect to the external world in which agents may be deployed:

- **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 pre-processed 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.

- **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 HTTP 304 (Not Modified) or 404 (Not Found) responses is an agent of the logical layer.

- **Syntactic** agents operate on structured or semi-structured input with *a priori* fixed meaning and schema. As opposed to a physical agent receiving 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 XML[2] data that may be exchanged with other agents capable of parsing and generating the given schema is another example.

- **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)[3]. 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.

- **Human Interface** agents are situated in an environment where they work in concert with a human user. Examples include "paper-clip" agents, interactive proof-checkers, CAD/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 above classifications are for single individual agents, but have implications for multi-agent systems as well. These are further discussed in Section 3.5.

3.5 Multi-Agent System 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. Properly designed, these interactions create much more substantial functionality than that of any single agent.

3.5.1 Dimensions of Multi-Agent System Complexity

The primary dimensions for classifying multi-agent systems are shown in Figure 3.8[4]. These axes position systems based on the number of agent instantiations they include, the internal complexity

[2] `http://www.w3.org/TR/REC-xml/`

[3] `http://www.w3.org/TR/owl-ref/`

[4] This figure adapted from the DARPA TASK and DAML Program briefings.

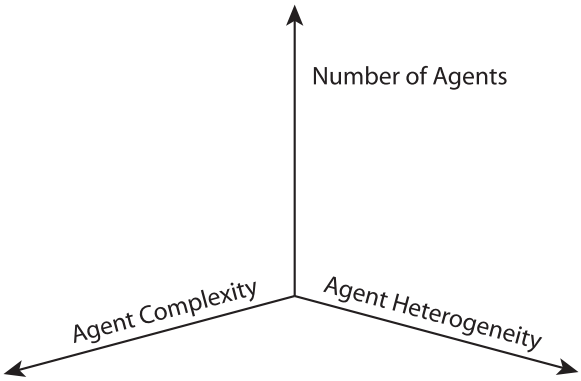


Figure 3.8: Dimensions of multi-agent system complexity.

Label	Heterogeneity	Number of Agents	Internal Complexity
Monolithic System	–	Single	High
Median System	Mid	Mid	Mid
Swarm System	Low	High	Low

Figure 3.9: Labels for common types of multi-agent systems.

of those agents, and the number of different types of agents. Classifications of internal agent complexity are discussed in Section 3.4. Based on these dimensions, the following terms are defined to describe common types of multi-agent systems. A tabular view of their positions on the axes of Figure 3.8 is given in Figure 3.9.

- **Monolithic System**. 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 artificial intelligence 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.

- **Median System**. Many multi-agent systems 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.

- **Swarm System**. A multi-agent system 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.

3.5.2 Structured Groups of Agents

A set of agents is collectively referred to as a **group**. In addition, a set of groups may also be referred to as a group. The term **multi-agent system** is used to denote a group of agents plus their supporting frameworks and infrastructure. A multi-agent system 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.

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:

- 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.
- 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.
- 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.
- 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 organizations groups within a society.

3.5.3 Communication in Multi-Agent System Layers

Multi-agent systems are comprised of several communication and interaction layers corresponding to the layers described in Section 3.2, as shown in Figure 3.10. 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 solid 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 intra-framework links provide for agents within framework instantiations to communicate.

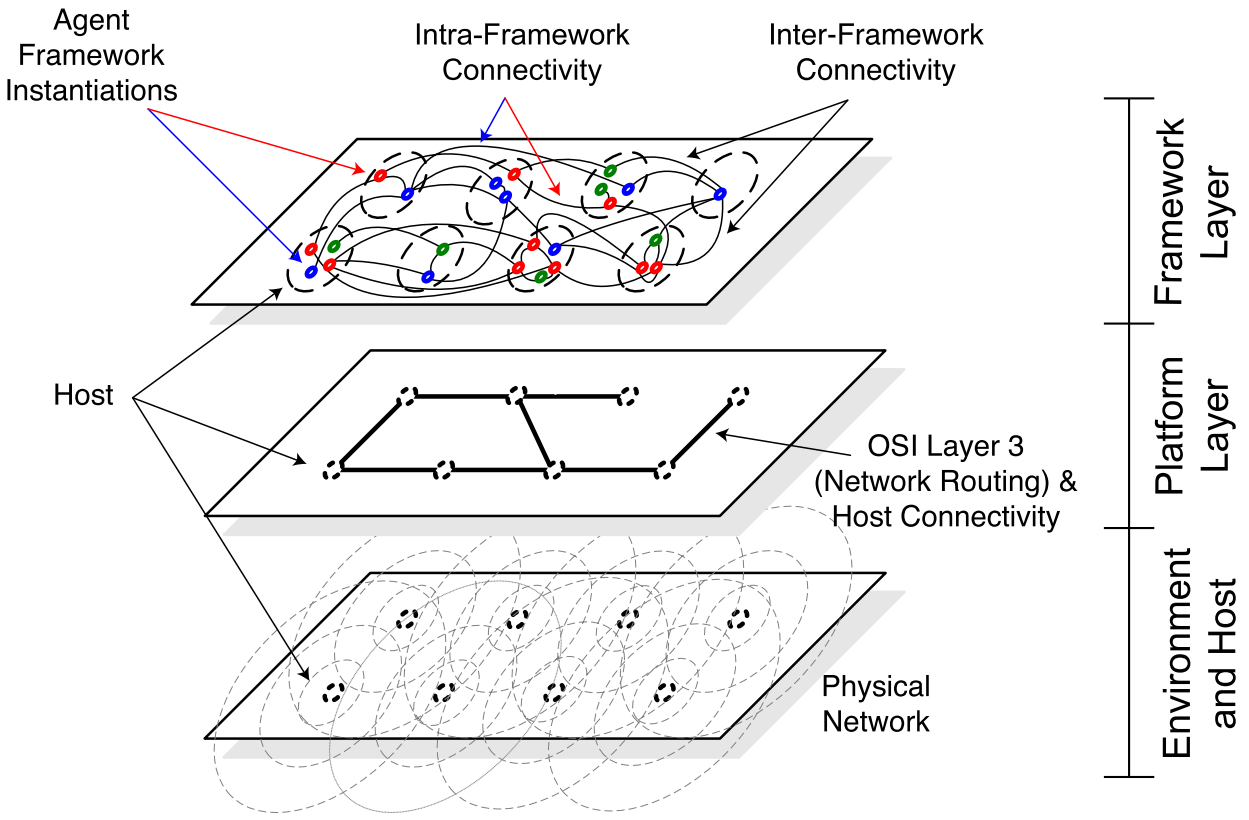


Figure 3.10: Communication may occur at multiple layers within an agent system. At each progressive layer, communication is rooted on and abstracts that of the underlying layers.

Chapter 4

Functional Concepts

This chapter 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 chapter 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.

4.1 Agent Administration

Definition: *Agent administration functionality a) facilitates and enables supervisory command and control of agents and/or agent populations and b) 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 below, the term "administrator" encapsulates all the administration functions although administration functions may not 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:

- **Agent Creation.** The act of instantiating or causing the creation of agents. In the example above, 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.
- **Agent Management.** 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.
- **Resource Control.** 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.
- **Agent Termination.** 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 those in saturated regions of the network.

4.2 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:

- **Authentication.** A process for identifying the entity requesting an action. Common examples include username/password credentials and use of public/private keys for digital signatures.
- **Authorization.** 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 that 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.
- **Enforcement.** 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; penetration technologies including viruses, Trojan horses, spoofing, sniffing, cracking, and covert channels.

4.3 Mobility

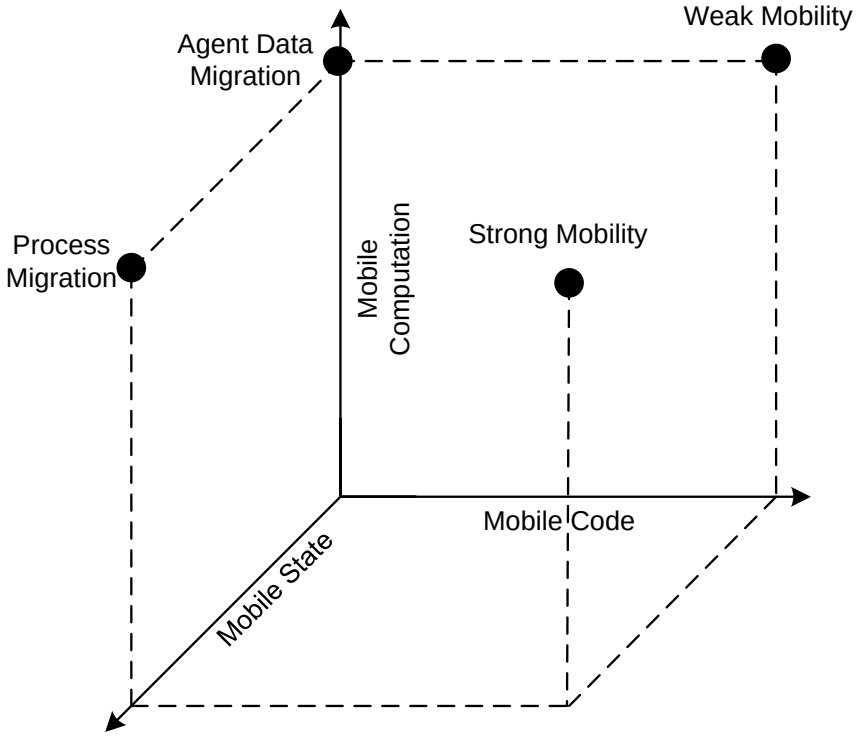


Figure 4.1: Axis of mobility features, adapted from [63].

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 more bandwidth-efficient for the agent to migrate hosts rather than to send the communications over the network.

As shown in figure 4.1, mobility capabilities exist along three axis. 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 computation support, *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 contemporary agent frameworks.

Process Model: Mobility functionality is described by the following processes.

- **Decision Procedure for Migration.** 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 independently decides when and where it migrates. 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.

- **De-register, Halt, Serialize.** Once an agent decides (or is notified) that it is migrating, it must de-register from all of the directory services on the framework instantiation with which it is 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 is the data members of the object. Some frameworks may support storing other information, such as the point at which execution stopped.

- **Migrate.** The process by which the serialized, non-executing agent leaves the source framework instance and arrives at a destination framework instance. This does not necessarily imply that the agent leaves 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, allowing for migration within a particular host. According to [73], 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. That is, an agent could serialize and "migrate" to itself. However, an agent system as a whole possesses mobility functionality if and only if it allows agents to migrate among different framework instances.

- **Deserialize, Re-Register 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. The agent re-registers 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.

4.4 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 multi-agent task planning language, such as TAEMS [53], 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:

- **Conflict avoidance** A process or mechanism for preventing conflicts. Examples of such processes include multi-agent planning algorithms (both on-line and off-line) that take care to produce action plans that do not have conflicts.

- **Conflict detection** The process of determining when a conflict occurs 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.

- **Conflict resolution** The process through which conflicts between agent activities are handled. 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, multi-agent planning, norms, social laws and teamwork models.

4.5 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 inter-process communications.

Messaging involves a source, a channel and a message. Optionally, a receiver may be designated, 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.

- **Message Construction.** The process through which a message is created, once a source agent determines it wishes 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.

- **Naming and Addressing.** 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.

- **Transmission.** The actual transport of the message over the channel. This may be a one-shot transmission or a continuous stream. One common model is messaging an agent on another host by 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.

- **Receiving.** 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, QoS and guaranteed delivery/timeliness.

4.6 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 providing informational, debugging or management information about the agent system as it executes. It can be a centralized service or distributed amongst 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.

- **Log Entry Generation** The process by which logging information 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 enters 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 through 5). The entry and any attributes are packaged into a data structure for writing.

- **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.

- **Accessing Log Entry** The logging functionality must provide a mechanism for a human user or 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.

4.7 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:

- **Naming.** 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 [72].

- **Notification.** 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.

- **Query Matching.** 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) [64].

Chapter 5

Software Engineering Methodology for Creating a Reference Model

5.1 Creating the Reference Model

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

- Capturing the essence of the abstracted system via concepts and components;
- Identifying software modules and grouping them into the concepts and components; and,
- Identifying or creating an implementation-specific design of the abstracted system

Reverse engineering and software analysis methods were employed to create the ASRM. By applying these methods, the process of creating a reference model is reversed. It is helpful and informative to employ reverse engineering techniques as a means of performing "software forensics" on existing (open-source and proprietary) agent systems and frameworks, due to the number of such agent systems currently available. By performing some analysis, one obtains the software modules that comprise the subject systems. Data is produced allowing the documentation and understanding of legacy software systems and for verification of existing software documentation. This data is further abstracted to obtain this abstract "essence" of the systems. By grouping, abstracting and querying this data in different ways, information is gleaned that simple observation may not find.

Reverse engineering techniques determine both the structural and behavioral makeup of software systems, including agent systems. The static analysis of the software system yields the structural components existing in the system, and the dynamic (behavioral) analysis shows how and when these components are instantiated and used. Moreover, behavioral analysis shows the runtime interactions between the components found during static analysis.

Reverse engineering techniques allow not only for identification of components within the reference model, but also to identify both structural and behavioral similarity 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. For example, through dynamic analysis it is possible to show only those components that are related to agent communication or migration. The result is a set of components that are mappable directly to the reference model, validating its relevance to existing agent systems.

5.2 Documenting the Reference Model: The 4+1 Model

According to [50], a comprehensive software architecture document provides a description of the software system using various **views**. Views are architecture descriptions of a software system in a particular context that is relevant to a group of stakeholders. Stakeholders may include developers, business-persons, customers, etc. Views illustrate system functional and non-functional requirements from various perspectives, and may overlap with one another.

The **4+1 Model** realizes the various types of component relationships that exist in software systems. For example, an inheritance relationship between components is not the same as a data flow or call graph relationship between components. Depending on the stakeholder, different relationships carry different weights and significance. For some, certain relationships are meaningless and can be disregarded.

The views presented by the 4+1 Model are:

- **Development View**: the package or development layout of the system;
- **Process View**: runtime behavior of the system, including concurrency relationships and ordered tasks carried out by components of the system;
- **Physical View**: the platform level view of a system, including servers and hardware requirements; and,
- **Logical View**: the static structural layout of the software system, including its object oriented design.

Because these views may overlap or be somewhat disjoint, there exists a view that summarizes all the other views in a cohesive way. This represents the "+1" view, called **Scenarios**. Scenarios use UML use cases to represent the interactions between the 4 views, and cross cuts them to aggregate the views into a software architecture. The remaining views are described in parallel with the use cases, and are depicted using other UML notations described in Section 2.2.

This model allows for streamlined documentation of the software architecture, and standardizes the metrics for measuring adherence to the architecture from various perspectives. Using the 4+1 model, Chapter 6 concentrates on analyzing systems from an aggregation and abstraction of these views.

5.2.1 Reference Model, Reference Architecture, Design and Implementation Hierarchy

In the use case diagram described in Figure 5.1, dependency arrows show that the implementation of a model derives from its design that derives from its reference architecture and that ultimately derives from our reference model. Actors show how they relate in the various stages of software development. A single reference model created by the agent systems community may drive software architects to create multiple reference architectures. In turn, one of these architectures may cause designers to create multiple system designs, each of which could have multiple implementations created by computer programmers.

Because the ASRM describes agent systems on the whole, and not specific implementation of agent systems, it is not appropriate to describe each of these four views independently of each

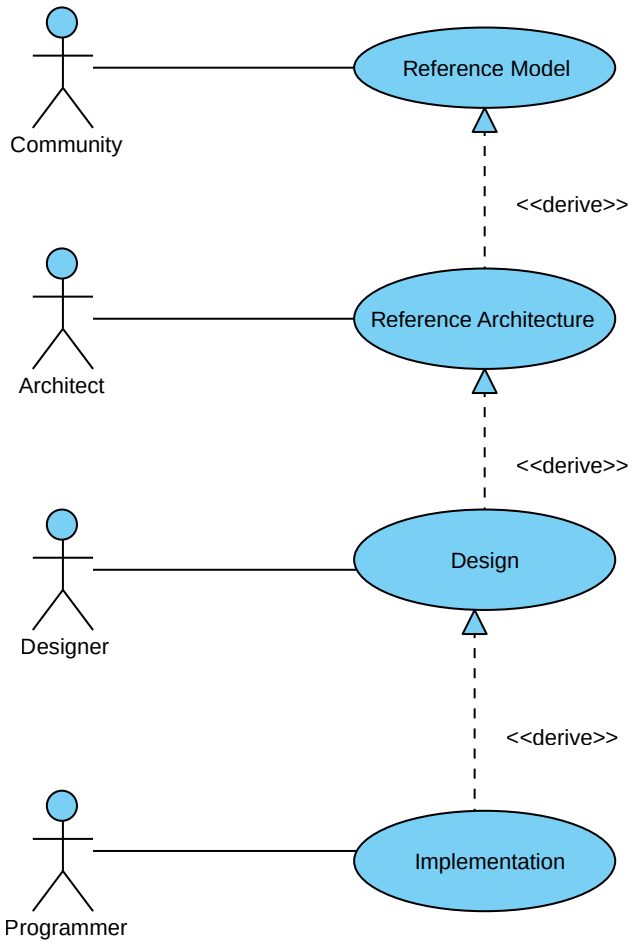


Figure 5.1: Hierarchy of Reference Model Abstraction

other. This is because the individual views describe specific systems at the implementation level, and not at the level of abstraction of a reference model. Instead, Chapter 6 uses aggregate views called "Structural Descriptions" (consisting of the Development View and the Physical View) and "Behavioral Descriptions" (consisting of the Logical View, the Process View, and the "+1" Scenarios View).

5.3 Reverse Engineering Techniques for Informing a Reference Model

Reverse Engineering (RE) is the analysis of software systems by extracting artifacts and functionality from an existing system. 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.

For example, 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 inter-relationships by identifying the level of coupling (the amount of relationships) and the type of relationships existing 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.

For example, an Operating System might contain a collection of components for handling graphical display, and a collection of components for handling disk operations. It is evident that these collections tend to inter-operate strongly amongst themselves, yet little interaction takes place between the collections themselves. Relationships that exist within a particular collection of components are called **internal relationships**. On the other hand, relationships that exist between collections of components are referred to as **external relationships**.

Collections of relationships, called **clusters**, are formed by grouping those components with 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 to 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.

This 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 as to a generic architecture (called a **reference**

architecture) that comprises a class of software systems, such as Operating Systems. In addition, RE validates and identifies discrepancies between that reference architecture and existing systems.

5.3.1 Static Analysis

Static analysis is the analysis of software using its source code as the primary artifact. The system needs 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, 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 abstracting a data repository for source code. 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.

5.3.2 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. Moreover, dynamic analysis can obtain data on behavior-specific aspects of the system such as threading and I/O, that 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.

Chapter 6

Structural and Behavioral UML Documentation of the Reference Model

This chapter illustrates the design, components, concepts and use cases associated with an "idealized agent framework," in which all common agent system components are implemented at the framework layer. While this is not an ASRM requirement for agent systems, it provides a simplified baseline for discussion. This idealized framework is depicted via a formal UML description of the reference model as a whole, using the UML subset described in Section 2.2. This example is called an "idealized" agent framework to highlight these assumptions for purposes of discussion, and to further illustrate that component implementation at any particular layer is not a requirement of the ASRM.

The goal of the idealized agent framework is to describe components and functionality that connects the agent(s) and the infrastructure. However, the ASRM does not preclude the implementation of those and other components within other layers of the system. In addition, it is possible that layers of the agent system, including the framework, are arbitrarily small or even non-existent.

6.1 Structural Descriptions: the Development View and the Physical View

The Development and Physical Views comprise this structural UML description of the ASRM. These descriptions use more abstract UML descriptions that traditional 4+1 structural descriptions to match the appropriate level of abstraction found in a reference model. These descriptions do not prescribe conformance to the ASRM; instead, an idealized example is provided including all functionality defined by the reference model, implemented at the framework layer.

6.1.1 Development View

Figure 6.1 and 6.2 illustrate a structural breakdown of components and subsystems to implement the functional concepts outlined in the ASRM. As mentioned, the ASRM does not prescribe actual object-level implementation detail of these components; for example, one may implement Security

Enforcement using RSA encryption, while another may take advantage of the operating system's file permissions structure. In this case, the ASRM does not even dictate the layer in which these components are implemented; moreover, a particular agent system may be a superset or a subset of these components. For the sake of discussion in a reference model, the implementation of these components is idealized and placed at the framework layer.

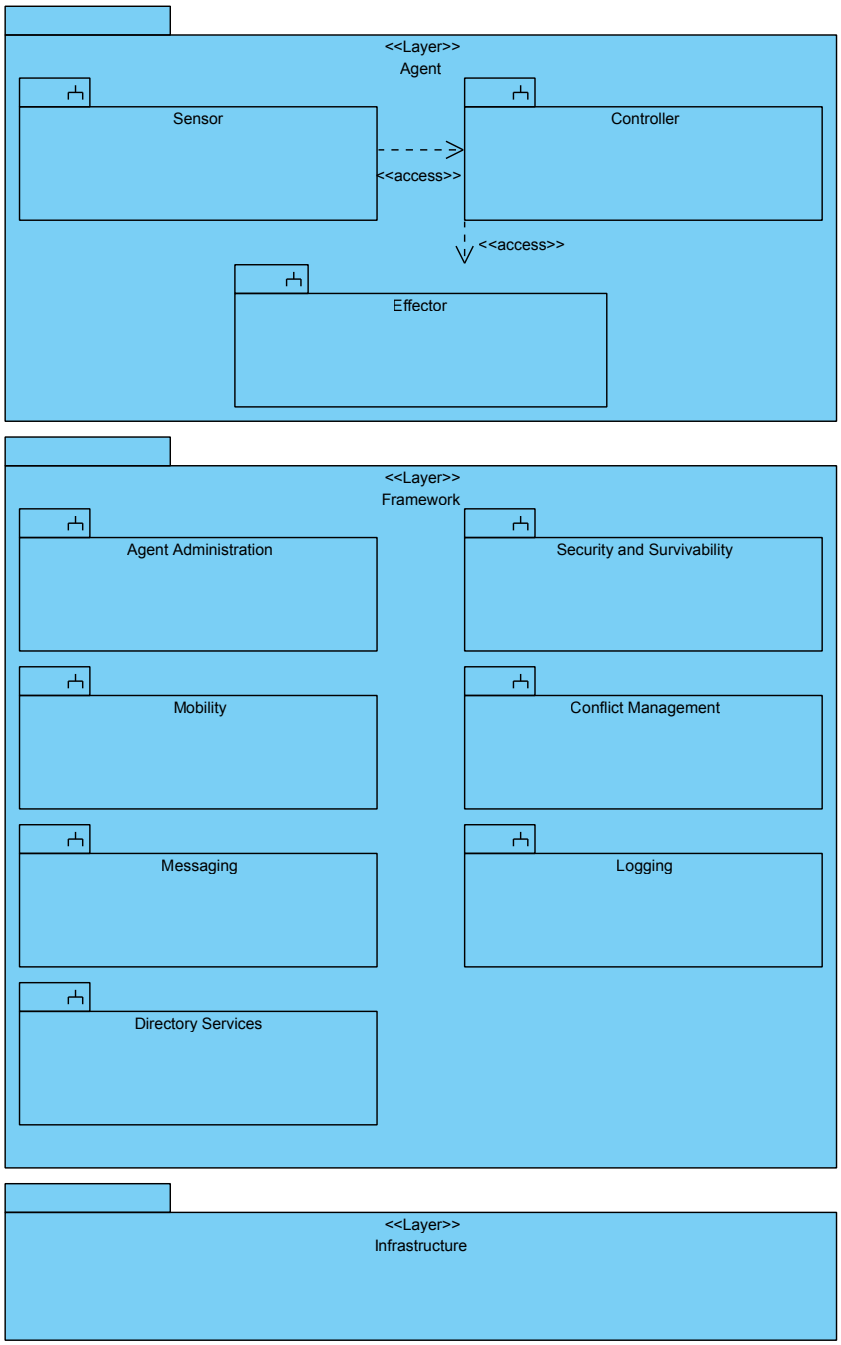


Figure 6.1: MAS Packages

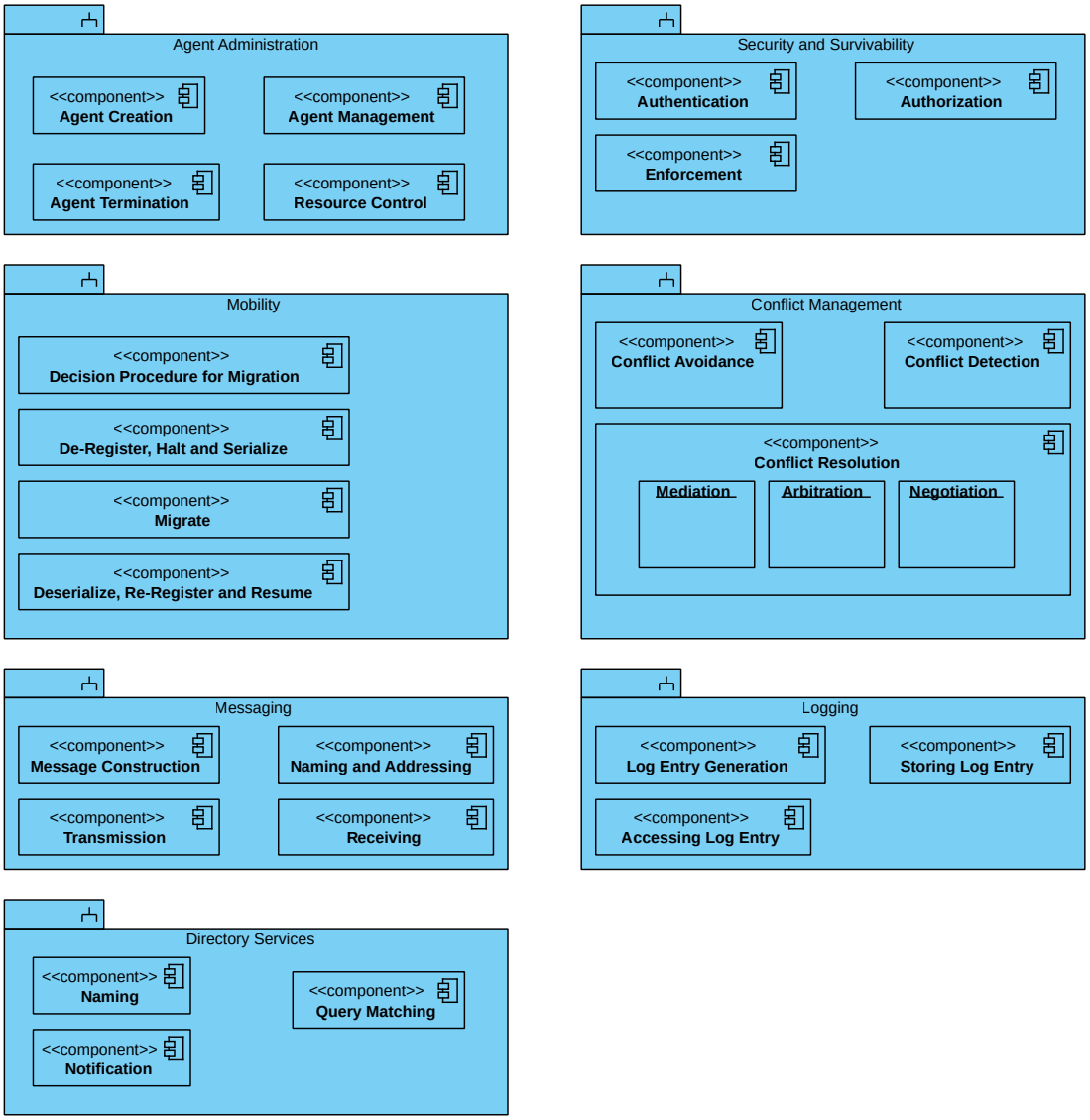


Figure 6.2: Framework Packages

6.1.2 Physical View

Whereas the Development View describes the functional concepts in an agent system, the Physical View describes the deployment layers also described in the ASRM. A functional description of these layers is given in Figure 6.1, and their deployment is shown in Figure 6.3. These layers are described in the ASRM, and allow for a heterogeneous suite of agents to exist in a physical environment. In this diagram, agents are grouped by their framework but may be physically scattered throughout a network.

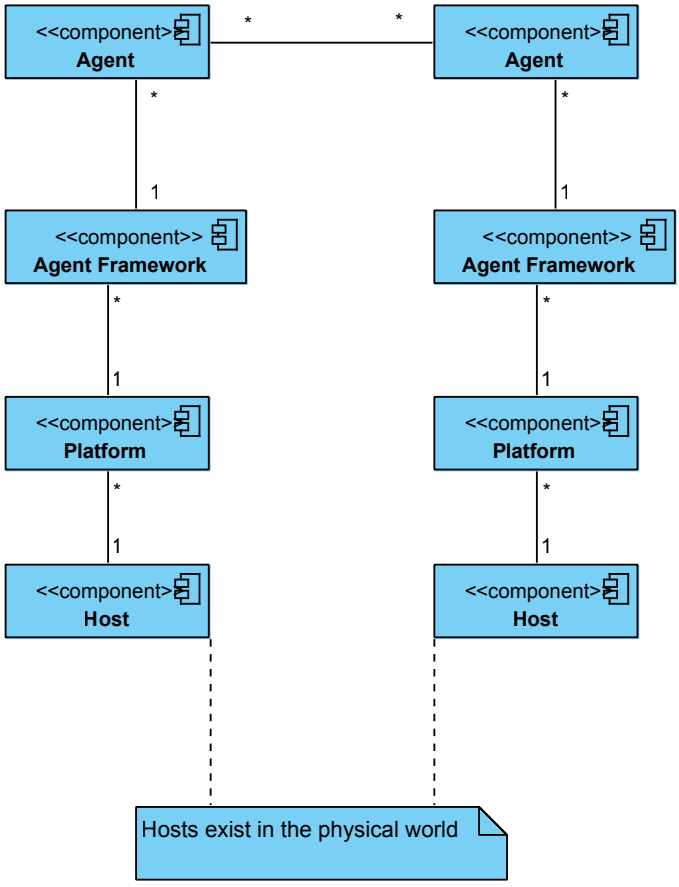


Figure 6.3: Agent System Layers

6.2 Behavioral Descriptions: the Logical View, Process View, and Use Case Scenarios

Complex computer systems require a standard method of documenting their purpose and design. UML provides such standard methods for describing this functionality from different perspectives

and levels of detail, including use cases, activity diagrams, and sequence diagrams. These three diagrams make up the UML behavioral descriptions, covering a coherent unit of functionality describing the interaction between an actor and a service provider. An actor is a human, computer, or another object. A service provider is one or more systems, subsystems, components, or actors. The purpose of these diagrams is to define a piece of behavior without revealing the internal structure of the behavior's implementation to various extents.

Use cases define the problem to be solved. These use cases are referred to as **scenarios**. According to [36], "a scenario is a sequence of steps describing an interaction between a user (the **actor** and a system." Actors are not just the users but also the roles that the user(s) take on. The system is the collective of components and concepts that exist therein. Activity and sequence diagrams define what is expected of the actors, a timeline of the actions taken and responses received, and any error conditions that may exist. Interactions between behaviors within the system are primarily illustrated as messages passed or relationships between the behavior and the actors.

Our scenarios are described through the following UML descriptions:

- **Use Case.** Use cases are diagrams that show high level processes that the system's functionality encompasses. They show interactions within the system and between users ("actors") and the system.
- **Activity Diagram.** Activity diagrams show system functionality as a business process model. Activity diagrams contain or elaborate use cases by assigning "swimlanes" to the actors. They show how the functionality of different system entities is related through decision cycles. Essentially, activity diagrams show interactions among the use cases.
- **Sequence Diagram.** A sequence diagram depicts system interactions in a more detailed way. Often, sequence diagrams illustrate the system at the object and method level; however, for purposes of the reference model, sequence diagrams elaborate upon the activity diagram.

For more information and a legend, see Section 2.2.

In the context of a multi-agent system, actors are not limited to human users, but the agents themselves. As a result, these use cases serve as a mapping between concepts within an agent system and the layers in which they might exist. As the ASRM does not mandate bindings between specific concepts and layers, many use cases may exist for the same function. These use cases present hypothetical and representative scenarios exercising the major concepts of an agent system in a particular way. The ASRM emphasizes for discussion the situation in which agent functionality is located at the framework layer; therefore, many of the use cases here illustrates this model.

6.2.1 Agent Society

As defined in section 3.5.2, a **group** of agents is any collection of agents. A number of terms describe groups of agents with particular properties. An agent **society** is a cultural grouping of agents based on a common set of laws, rules or policies. An **agency** is a more fine-grained group of agents that specializes in expertise or enabling a service in a given domain. An agent **team** is a group of agents that share a common goal or goals.

Each of the processes described in subsection 7.2.2 is an agency consisting of similar agents interoperating towards their objective. Each agency contains other agencies that comprise the

phases of the subprocess they represent. The agents are fully interoperable within the agency and across agencies.

These processes are abstracted to generic interoperable agencies to illustrate how these processes can be implemented by a multiagent system within the context of the ASRM.

6.2.2 Initializing an Agent System

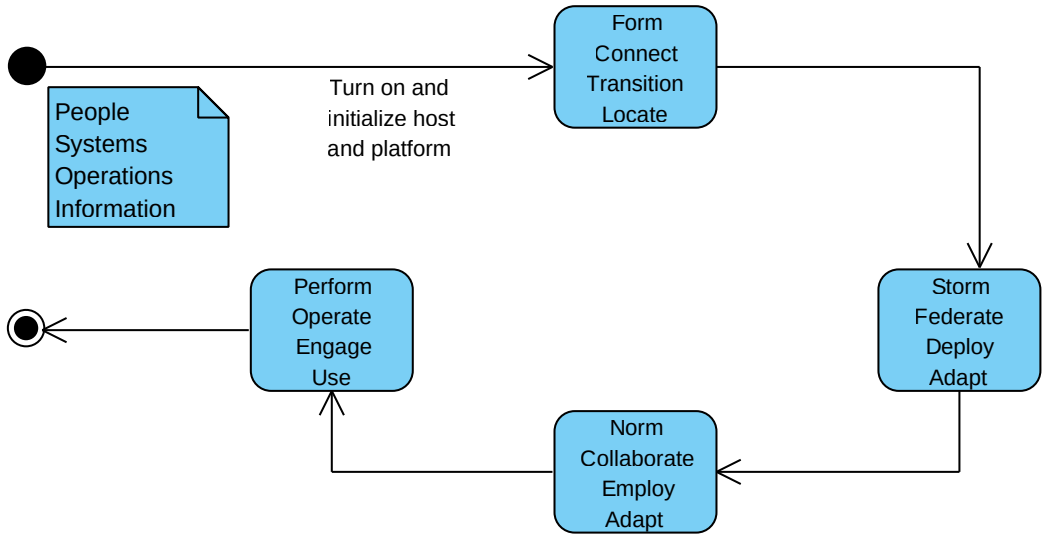


Figure 6.4: Flow of Teams, Systems and Mission with Data.

Interactions within an agent system follow the Tuckman and Jensen[66] "Form, Storm, Norm, Perform" paradigm for team development (Figure 6.4). The original Tuckman paradigm contains four stages, as follows:

1. Form stage: the group begins to take hold but is generally in a state of initialization.
2. Storm stage: the group operates without an agreeable protocol and conflicts generally need to be resolved.
3. Norm stage: the protocols are established and a plan is made based on the group's abilities.
4. Perform stage: full operation takes place.

A fifth stage, the **Adjourn stage**, was later added to illustrate the normal termination of the group's operation and/or existence.

Following this paradigm, and as illustrated in Figure 6.5: the host and platform are invoked, the framework is instantiated, and the framework instantiates and initializes the agents. Once this is complete, any number of agents are spawned and grouped into teams based on their goals, agencies based on their function or societies based on their observance of a common protocol. At least one human is part of this process for purposes of overseeing the activity, and inputting or modifying the

mission for the agents. Once the mission is given, the agents can further divide into other groups and begin their decision cycles and information exchange to work towards their goals.

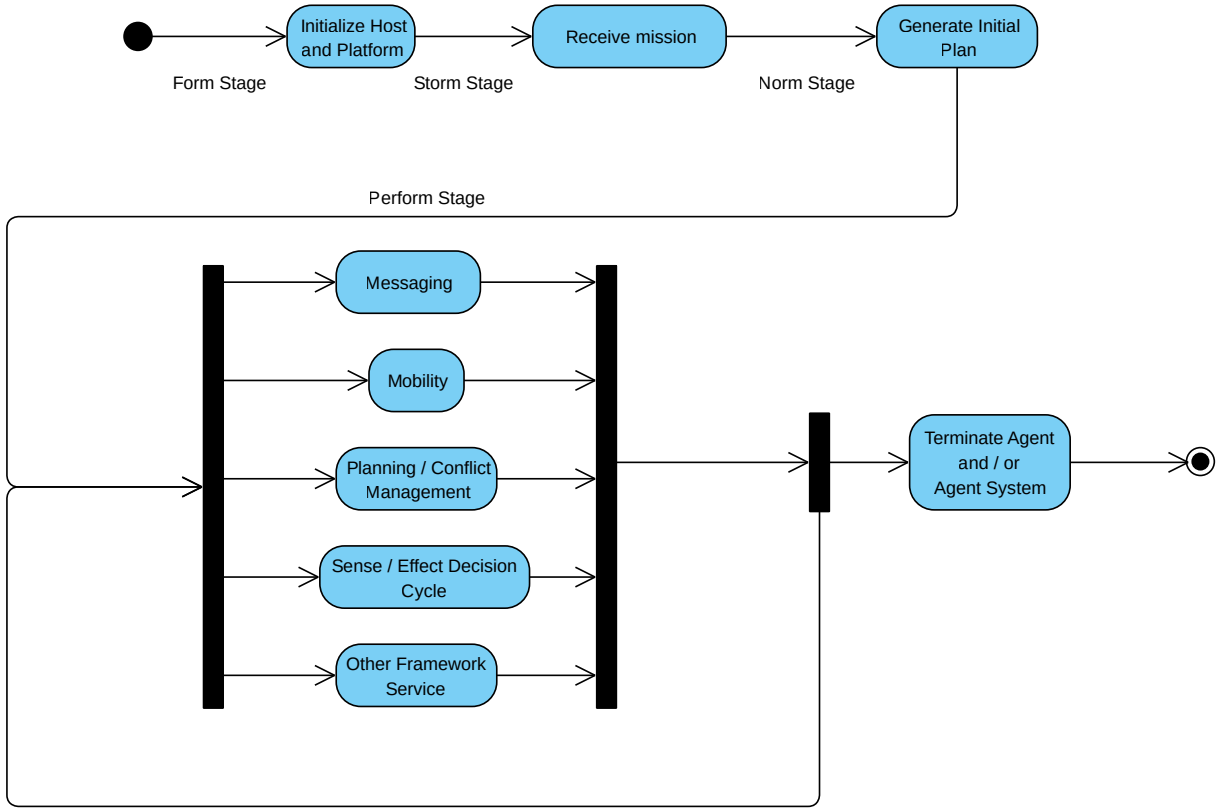


Figure 6.5: Initialization of an Agent System.

Chapter 7

Mapping Existing Systems to the Reference Model: Case Studies

7.1 Agent Framework Mappings to the Idealized Framework

7.1.1 Scenario

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 those 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 that s_1 resides on 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 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 presented first, followed by an in-depth analysis highlighting components and tracing execution.

Execution of this behavior is traced using the EJP tool. From these traces, one draws conclusions about the framework's architecture and make mappings to a reference architecture (and, thus, to the reference model). Several figures are included in the next few analysis sections. These figures show the raw output of EJP. The run-time 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.

Nodes are colored using a red spectrum where the brighter reds indicate more time spent executing that particular function. 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.

7.1.2 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.

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. In Figure 7.1, the Platform is instantiated followed by an `AgentContainer`. 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 Figure 7.2. The general procedure for a migrating agent seems to be 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 and appears as in Figure 7.3. Again, the agent does nothing and is cleaned up by the agent framework instantiation in the `agentFinished` function.

With regard to message passing, it is assumed there are two agents on the same framework instantiation. The first agent is created as in Figure 7.4. Similarly, the receiving agent, is depicted in Figure 7.4. Looking closely, one sees the where the message is created; however it is not evident where it is received. This is explained in the following Mapping section.

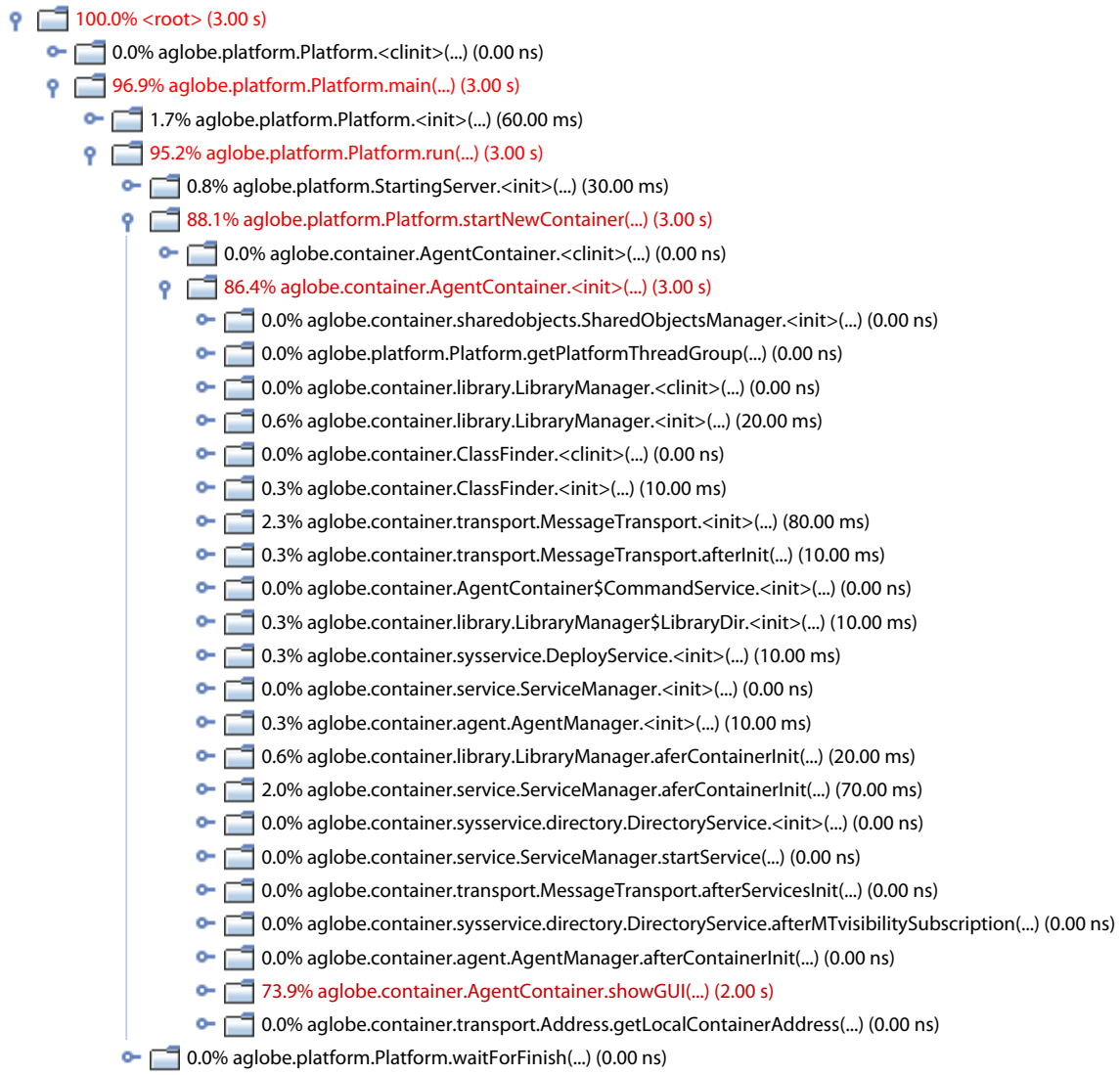


Figure 7.1: A-Globe framework dynamic analysis data. This diagram illustrates the initialization of the A-Globe framework and instantiation of a sample agent. Initialization of the A-Globe communications library, Agent Manager, and Directory Service are shown here. Note that A-Globe refers to itself as the "platform," whereas the ASRM considers this to be the "framework." This is a difference in nomenclature only.

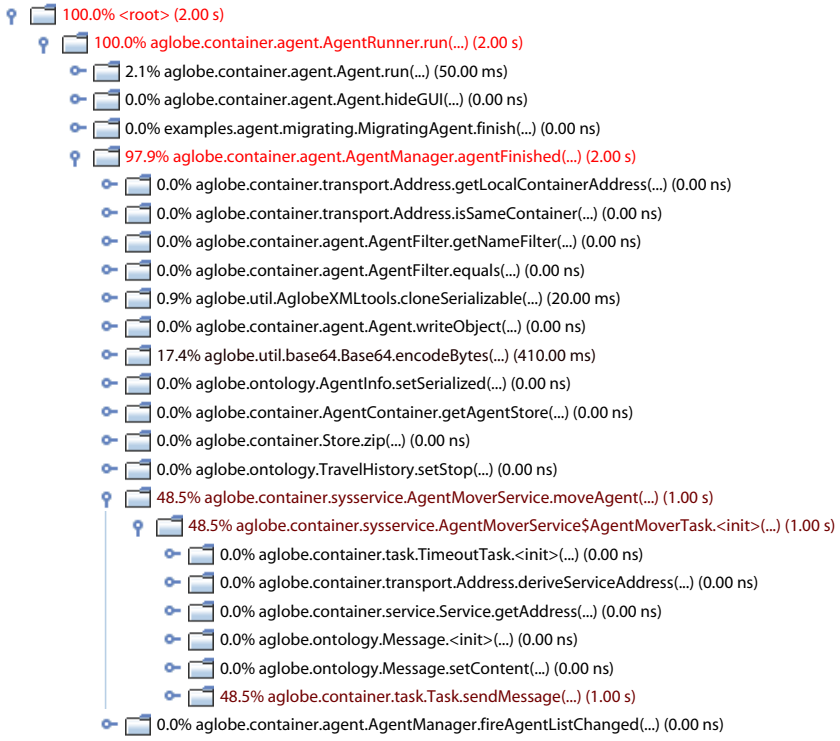


Figure 7.2: A-Globe migrating agent, *m*, before migration. A-Globe agents migrate using a procedure consistent with the ASRM mobility functional concept.

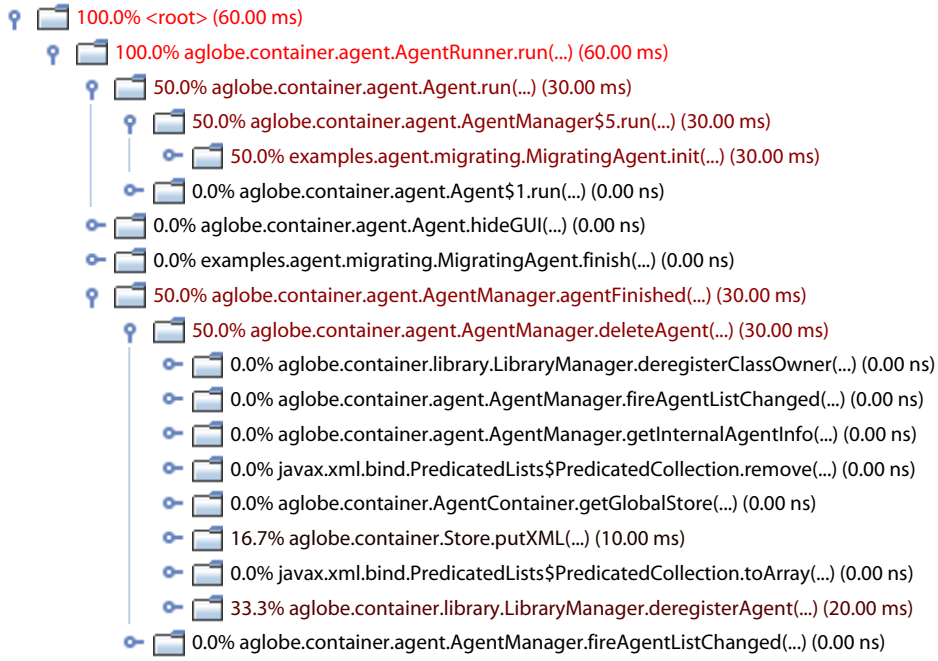
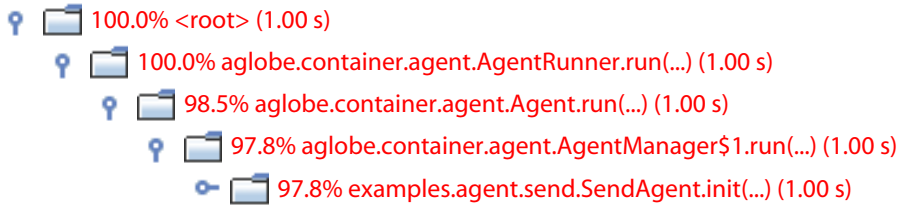
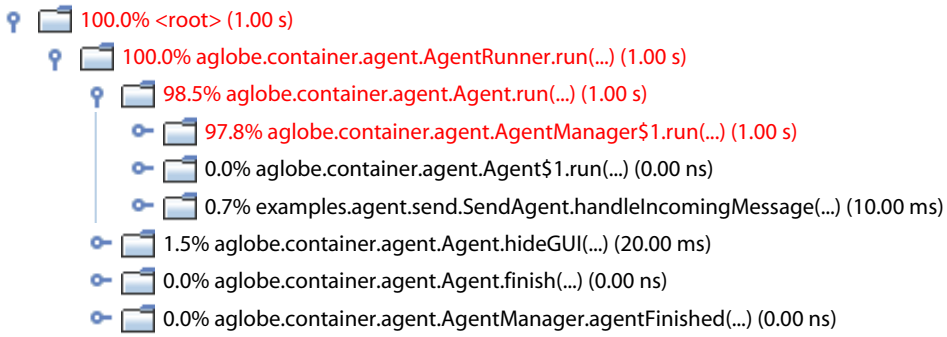


Figure 7.3: A-Globe migrating agent, *m*, after migration. As seen here, in A-Globe the serialized agent is de-registered from the source framework instance after the agent is deserialized on the destination host.



(a) The Sender Agent (s_1) Initialization



(b) The Receiver Agent (s_2)Initialization

Figure 7.4: A-Globe sending agent s_1 and receiving agent s_2 dynamic analysis data.

Mapping to Idealized Agent Framework

An examination of the A-Globe 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. As shown in Figure 7.1, 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.

The agents also act in accordance with the idealized agent framework. 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 Figure 7.2. When the migration agent arrives at the second framework instantiation, it is re-created by the `AgentManager` and executed. Again, it does nothing and is then deregistered from the directory terminated.

The creation of the message is not shown in Figure 7.4 because of the abundance of noise generated by the GUI combined with unsuccessful attempts at filtering it. There is strong 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 that is shown in Figure 7.4. This experiment also exercised the logger because the message was properly displayed on the logger of the receiving agent.

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.

7.1.3 Jade

The Jade 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 performs the same task on new static agents s_3 and s_4.

Overview

The Jade framework is composed of several classes, including Boot and the entire `jade.core` package. In particular, Figure 7.5 uses containers similar to A-globe. 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`. Analysis of the static agents in Jade shows the control flow for message passing. All of the static agents are the same and appear as in Figure 7.6.

Before and after snapshots of the migrating agent are shown in Figures 7.7 and 7.8, respectively. In the before snapshot, the static agents are created `(TerminatorAgent.setup)` and the agent is then cloned and sent to the other host. The after snapshot shows how the agent is regenerated and creates the two static agents.

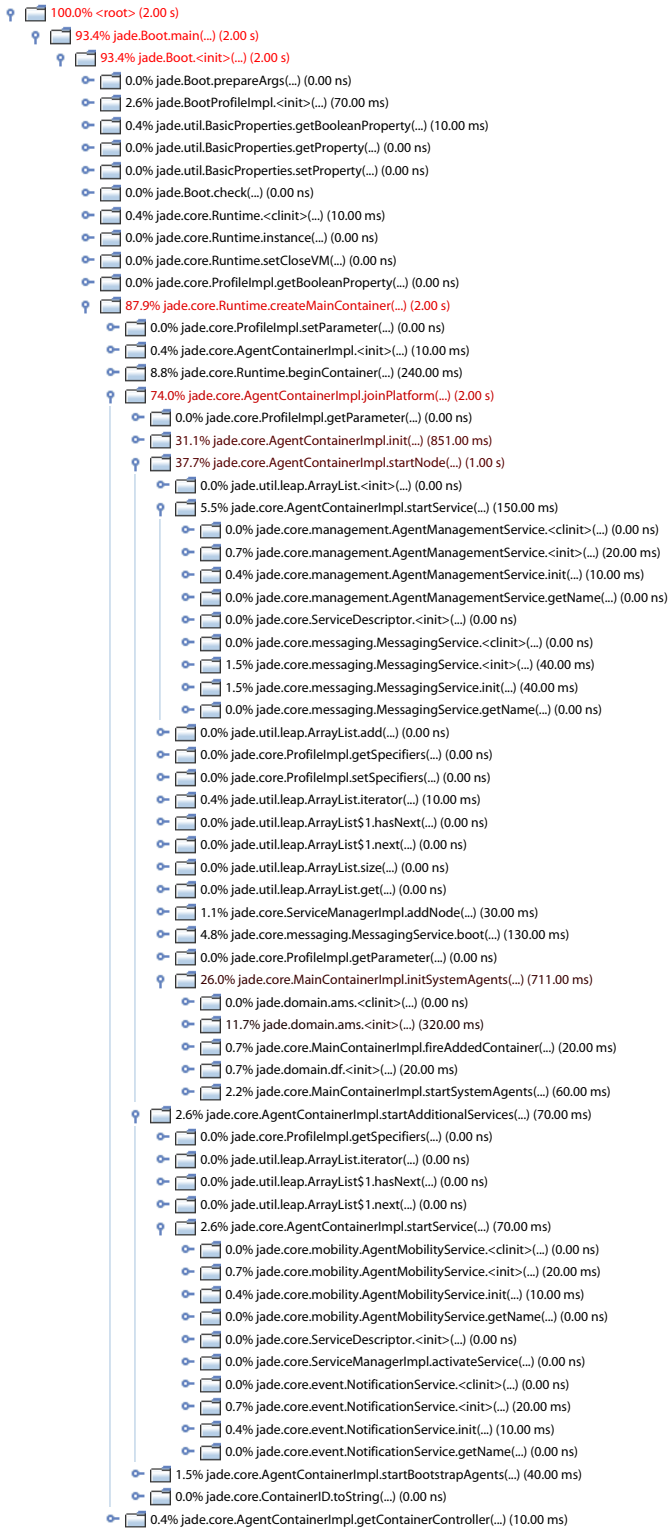


Figure 7.5: Jade framework dynamic analysis data. Here, the `AgentManager`, Messaging Service, and Mobility Services are instantiated as a part of the framework.

Figure 7.6: Jade static agents dynamic analysis data. Here, the agent is initialized and run by the framework.

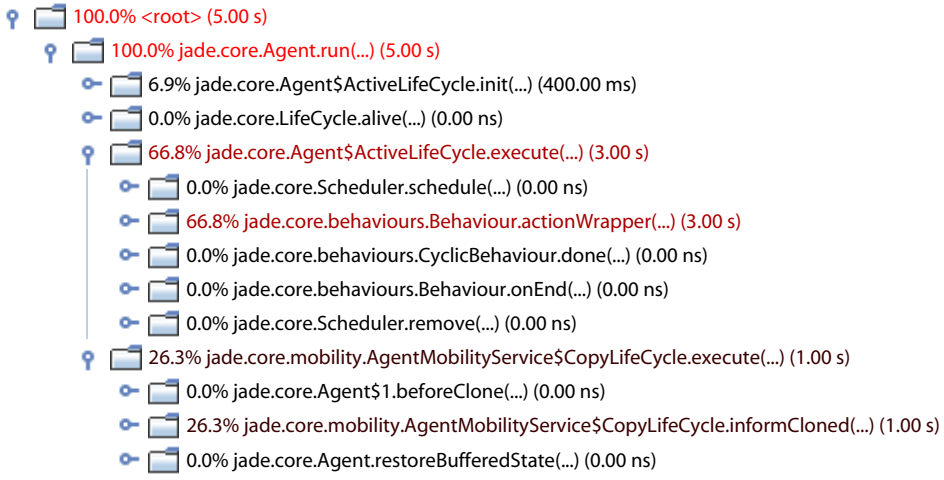


Figure 7.7: Jade migrating agent *m*, before migration, dynamic analysis data. The agent is cloned, copied to its destination, and restored.

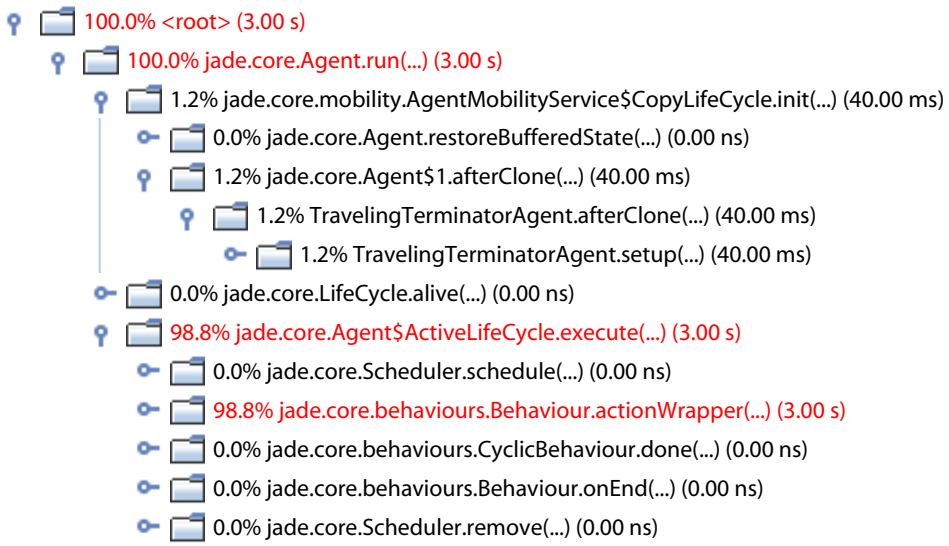


Figure 7.8: Jade migrating agent *m*, after migration, dynamic analysis data. This analysis provides more detail about the restoration process when the agent is deserialized.

Mapping to the Idealized Agent Framework

In Jade, the environment and host layers are implicit while the JVM represents a portion of the platform. When the Jade framework is started, a `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 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 that is expanded in Figure 7.5.

First, message passing is examined in detail. Two `JadeCommunicationAgents` are created that send messages to each other and then terminate. In Figure 7.6, the static agents take advantage of Jade's planning language that includes `Behaviours` 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 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. First, in Figure 7.7, 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. In Figure 7.8, the after snapshot is available. 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. 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.

7.2 Case Studies

7.2.1 Command and Control (C2)

The Command and Control Battle Command Information Exchange model is an example implementation of multi-agent agencies and societies.

Agencies within a society share a common objective, even if their individual goals or means are different. The information received is often the same or similar, and represents the high level objective. The agencies asynchronously digest this information and internalize it.

In the Command and Control Information Exchange context, information is first located during the form stage. It is not processed or interpreted, but simply acquired. It is stored for processing during the storm stage, and then adapted to meet the group's needs during the norm stage. Once this is established, the information is used, aggregated and turned into knowledge, experience, and action during the perform stage.

This process occurs in cycles in the context of several Battle Command processes (agencies). The communications process of Connect, Federate, Collaborate, and Operate follows the Tuckman paradigm (see Figure 6.4). Each process performs this cycle asynchronously, applying and

sharing aggregated information as needed. These domains include: Intelligence Preparation of the Battlefield (IPB) for analyzing the enemy and logistical information such as weather and terrain, Command and Control (C2) for administration of the operation, Decision Making (DM) for planning and management functions, Targeting (TGT) for operational control, and an omnipresent Information Exchange (IE) that supports all the domains with common objective information. See Figure 7.9.

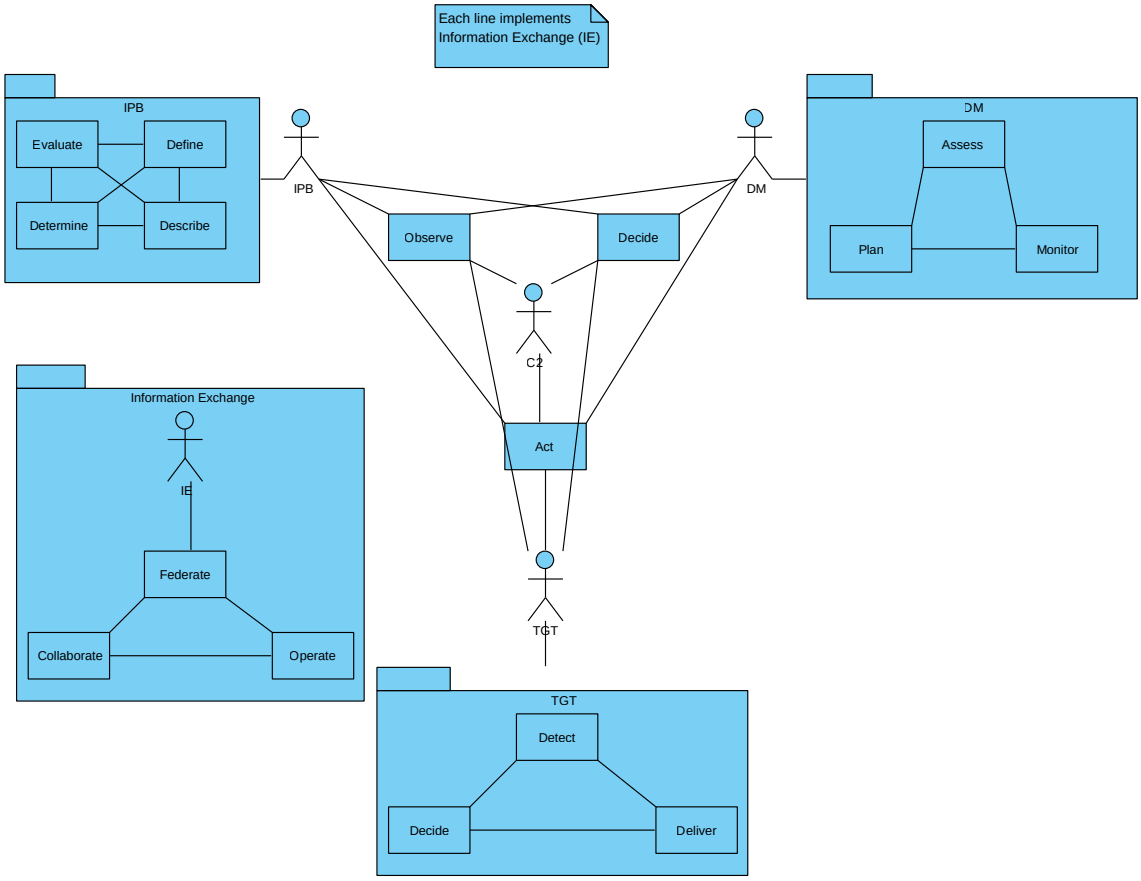


Figure 7.9: Battle Command Information Exchange.

In each of these agencies, the goal is to turn raw data obtained during the form stage into experience and action in the perform stage. This information is often shared with other agencies in an aggregated form. As shown in Figure 7.9, the asynchronous "output" of one agency might be an input for another. The aggregation and domain specific processing of this data adds value and efficiency to the society (see Figure 7.10).

In Figure 7.9, each subprocess of C2 is a process itself. These subprocesses are interconnected and interoperable. For example, the C2 process is made up of "Observe," "Decide," and "Act," and each of these corresponds to Intelligence Preparation for the Battlefield (IPB), Decision Making (DM), and Targeting (TGT), respectively.

These Command and Control processes represent specific implementation possibilities for agent communities within the ASRM. However, a more flexible and intelligent design allows for

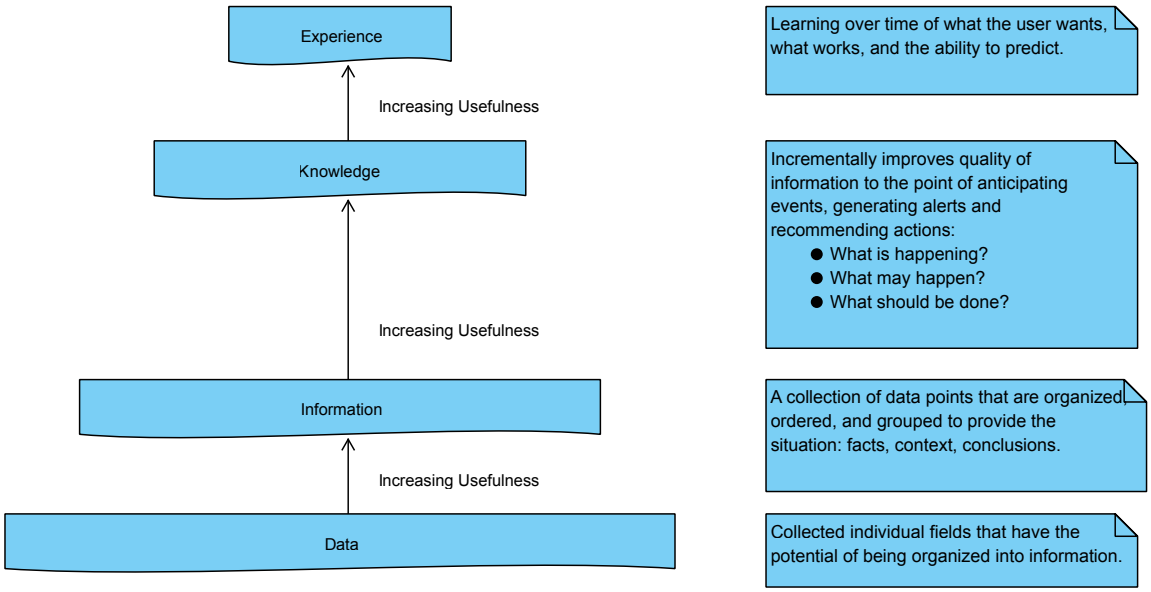


Figure 7.10: Hierarchy of Domain Resources.

a more generic and abstract agent implementation that is independent of the domain process being conducted.

For example, the X2 Intelligence officer manages the IPB process previously described. Generally, each officer has a process to manage. Regardless of the process, there exist behaviors (like the Information Exchange process) that dictate how these officers and their processes interact.

These processes are described as a case study for a possible implementation of agent societies and agencies within the context of the ASRM.

Intelligence Preparation for the Battlefield. Intelligence Preparation for the Battlefield (IPB) is an implementation of the Operational Net Awareness (ONA) process. Its product is a document of knowledge and intelligence that an X2 officer gives to an X3. The subprocesses within IPB show how intelligence officers think about actionable intelligence.

The C2 Observe process delegates to and oversees the operation of the IPB subprocess. The IPB phases (subprocesses) of "Define," "Describe," "Determine," and "Evaluate" also interoperate with one another to produce their result.

Decision Making. Similarly, the Decide subprocess of C2 oversees the Decision Making (DM) subprocess. This subprocess includes phases "Assess," "Plan," and "Monitor." As before, all subprocesses interoperate and produce a plan under the supervision of the X3 officer.

Targeting. The Act subprocess of C2 corresponds to the Targeting (TGT) subprocess. The TGT subprocess consists of the "Detect," "Decide," and "Deliver" phases that interoperate with one another.

This process is a good illustration of the modularity of the C2 subprocesses, because it is clear that this Act subprocess is not necessarily implemented by Targeting. Instead, any actionable process is used as long as it accepts a plan from the Decision Making process and collaborates with the IPB process.

Information Exchange. As seen in Figure 7.9, the processes, subprocesses and phases of Command and Control "interoperate." In fact, this is accomplished through information exchange and is illustrated by the Information Exchange (IE) subprocess.

However, as in human communications, this is not a simple matter of sending information; a protocol must be followed to establish a conversation and send the information in an orderly and expected way. Moreover, effective information exchange must take into account both the IE Requirements and IE Desirements of the recipient.

The ISO OSI Reference Model provides the process by which this information exchange occurs. C2 implements this process via the Tuckman paradigm and is consistent with the Messaging component of the ASRM. C2 Information Exchange includes the following phases:

- **Federate (Connect).** This represents the low level ISO communication layers in which a connection is established between the communicating parties. No data is exchanged, but the communication channels are simply opened. This process is independent of the environment or medium to be used for communication; this information exchange can occur over telephone wires, a network, written correspondence, etc. This Federate phase corresponds to the Tuckman form stage.
- **Federate (Initialize).** This represents the higher level ISO communication layers in which a protocol is chosen and the communications channels are prepared for high level message passing. This Federate phase corresponds to the Tuckman storm stage.
- **Collaborate.** Data is exchanged but no action is taken. The exchanging parties must discuss and interpret the exchanged information before taking action on the information. The Collaborate phase corresponds to the Tuckman norm stage.

- **Operate.** Finally, action is taken and the exchanging parties interoperate based on the information shared. The Operate stage corresponds to the Tuckman perform stage.

The IE subprocess is omnipresent and domain-independent. As such, this process is used by all the other processes within C2.

7.2.2 Mapping the C2 Domain to the Civilian Domain

The C2 domain described is mappable to the generic civilian domain by investigating civilian business operations. As shown in Figure 7.11, teams for accomplishing a mission are broken down into three categories:

- **Organization**: The organization is responsible for carrying out the broad mission that reflects what the entire organization is primarily responsible for. Civilian businesses describe this in a mission statement; the military describes this in terms of broad national defense goals.
- **Support**: Support units provide specialized improvements to the process of accomplishing the mission. For example, a better assembly line and the manufacturing of parts are examples of civilian support units. Artillery or air units are examples of military support; they are specialized and have specific goals towards making the mission easier to accomplish; however, members of the organization are responsible for ultimately achieving the mission.
- **Service Support**: Service support units are similar to support units except that they provide logistical support. Fuel, facilities and transportation services are examples of service support units in both domains.

These are called "Lines" in the civilian domain and "Combat" in the C2 domain, but their functionality is similar. It is important to note that these terms are defined loosely and are relative to the domain. For example, an organization that provides business facilities are considered a Service Support unit to the businesses that utilize those facilities. However, employees of that provider are considered units within the Organization. The goal is to assign agents to assist and adapt to each of these groups to aid in the production and execution of Business Products.

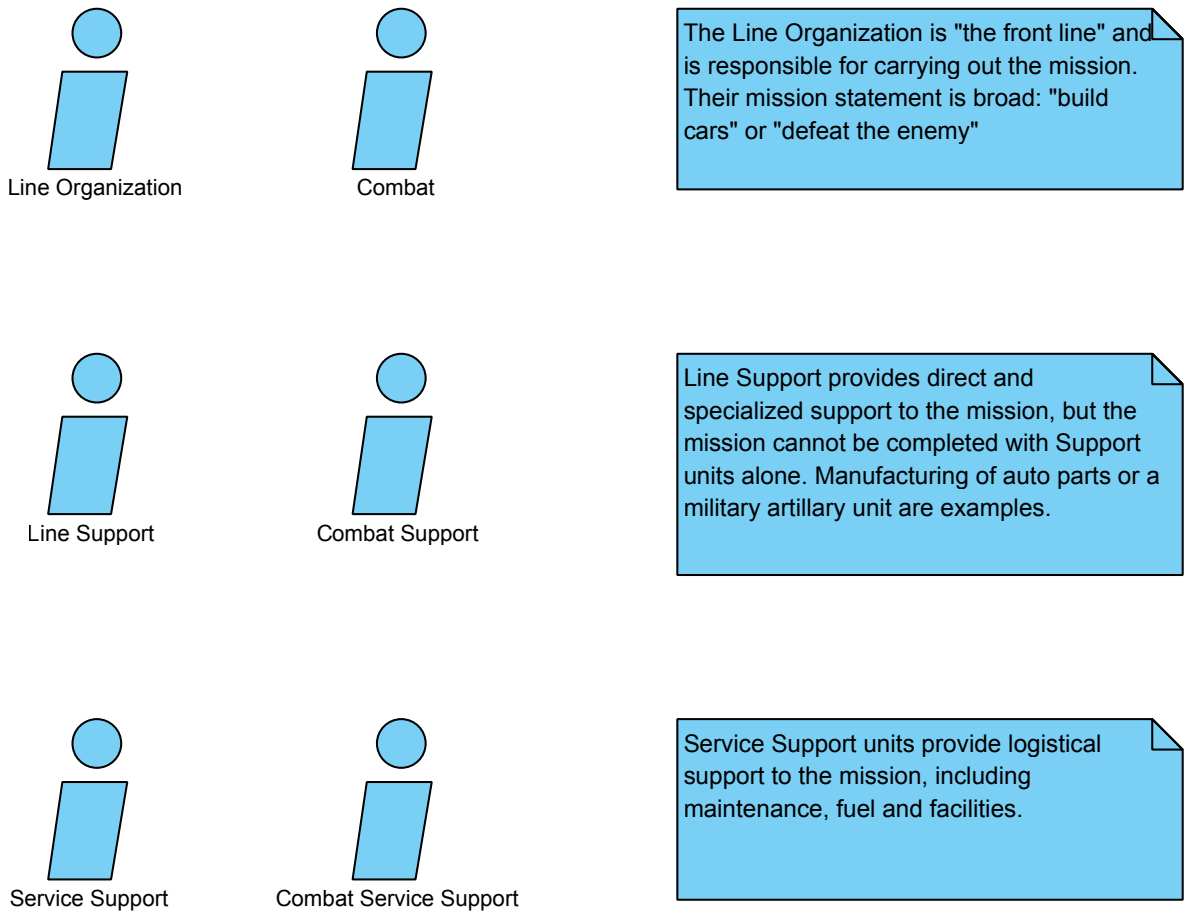


Figure 7.11: Correlation Between C2 and Civilian Domains

Abstract Workflow Processes. As an example, consider an agent framework "toolkit" providing basic functionality to support the Command and Control process. This toolkit is described in Figure 7.12.

In this toolkit, agents are responsible for these processes, under the direction of other agents and human intervention. Processes are modular and adaptable to each tactical scenario. Also, the agents depicted in this abstract workflow toolkit and in the example case studies that instantiate it refer to either software agents, hardware agents, or human actors. Regardless of the exact behavior of the process, the agents that control the process are able to intercommunicate and interoperate. It is important to note that no prescription is made as to the exact implementation of these agents, nor that they even share a common framework. If the heterogeneous agents refer to a similar high-level protocol for communications and interoperability, as is described by the ASRM functional concepts, this interprocess communication is possible. This makes the agents reusable and extendible.

C2 Products: Business Objects. As in the civilian model, Command and Control processes produce products or business objects through the hierarchy of command. See Figure 7.12.

In the C2 domain, these business objects include Orders and other Reports conducive to information exchange among the users and processes. At a higher level, the military joint Command, Control, Communications, Computers, Intelligence, Sensors and Reconnaissance (C4ISR) exchanges different but similar pieces of information both within and between branches of the military (for example, the Army and the Navy).

Through the ASRM, it is possible to establish teams of agents that are highly modular via the workflow toolkit described in Figure 7.12. However, it is ideal if the business products produced were also modular according to a common schema such as XML. This promotes ontology sharing and information exchange between societies – in this case, between branches of the military. Such a schema and a discussion concerning its benefits towards a C2 product-centric information exchange process is found in [55]

In the abstract case, a product might be described in an XML schema including the top level product containing a number of topics. Each of these topics contains components comprised of facts. A fact is a statement in the form of a "W6H" tuple ("who does what to whom, when, where, why, and how"). This structure parallels that of a book or other document, containing chapters, paragraphs and sentences. By modeling facts as W6H tuples, it is clear that they are similar in nature to a standard sentence.

These products contain information in an aggregate form relevant to the goals and mission of higher level command. In the C2 context, they convey TTP (Tactics, Techniques, and Procedures from Doctrine and Experience) data, information, and knowledge (see Figure 7.10). Depending on the context and source, however, this product may take on slightly different forms. Therefore, it is imperative to have a common schema that allows agents of any agency to interpret products of another agency; this allows for complete modularity not only of the agents but of their products as well.

Example C2 Business Object: OPORD. An example of such a product is the C2 OpOrder (OPORD). The OPORD is a product containing the following topics and some example components within them:

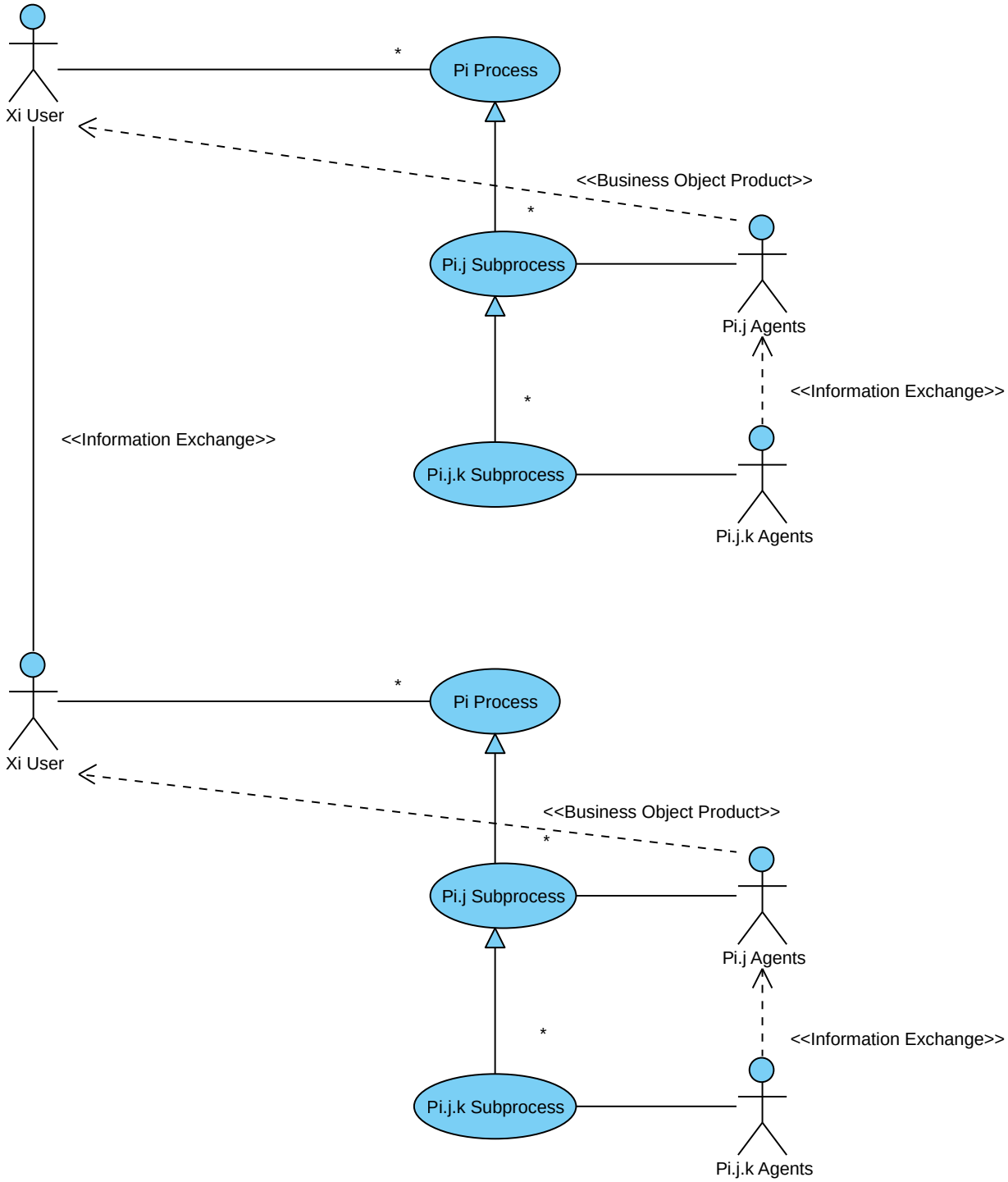


Figure 7.12: Abstract Workflow Process

1. Header
2. Situation
   (a) Enemy
   (b) Friendly
   (c) Civilians
   (d) Terrain
   (e) Weather
3. Mission
4. Execution
   (a) Cdr Intent
   (b) Ops Concept
   (c) CbtTasks
   (d) CbtSptTasks
5. ServiceSpt
6. C3

A. TaskOrg Annex
B. Intel Annex
C. OpsOverlay Annex

Each paragraph and annex of the OPORD represents a specific group of individuals as described in the C2 and business domains. The **Situation**, for example, represents the Organizations mission statement. The Intelligence analyzed and placed here are forms of Line Support. Via the ASRM, agents can assist in the production of such Situation reports from the available intelligence.

Within the Situation section are subsections for **Enemy**, **Friendly**, and so on. These sections are produced via the efforts of Xi officers specific to the C2 domain, described in the Abstract Workflow Toolkit in Section 7.2.2. In particular, an S2 officer is responsible for **Enemy**, **Civilian**, **Terrain**, and **Weather**, while an S3 officer is responsible for the **Friendly** section. In fact, each section of this document is produced in this way. It has been discussed that agents can provide support to Xi officers producing Business Products.

This breakdown further introduces a chain of command in which agents could assist and interoperate. Annex A describes the **TaskOrg Annex** and discusses the types of personnel available, what they do, whom they support, and which other personnel they can interact with if their current commander is busy, etc. This type of activity is domain specific but agent systems could assist a human in the decision process.

7.2.3 Agent Society Example: Integrated Process Team (IPT) Structure

The System of Systems Integration (SoSI) serves as a society with a number of agencies specializing in particular domains. Other examples of societies might be CERDEC or Tank Command. Their objectives are broad and can be subdivided into agency level goals. They are cultural in nature, sharing common objectives, while the agencies are functional partitions of the societies, sharing common ontologies.

In this example, the agencies within the SoSI society include the Integrated Process Team (IPT). The IPT is a functional group adapted from the industry context to achieve consensus within a domain. This group is made up of members of government, industry, and academia. IPTs are organized in a hierarchical fashion such that some IPTs have subIPTs. For example, the Networking IPT is the parent of the Intelligent Agents subIPT that contributed greatly to this document.

Interaction and information exchange between the subIPTs, their parent IPTs and the SoSI society mirrors the information exchange paradigm described in Section 7.2.1.

7.2.4 Situated Agent Example: Robot Soccer

Agents are typically situated within an environment and are able to interact amongst 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.

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 Figures 7.13, 7.14 and 7.15.

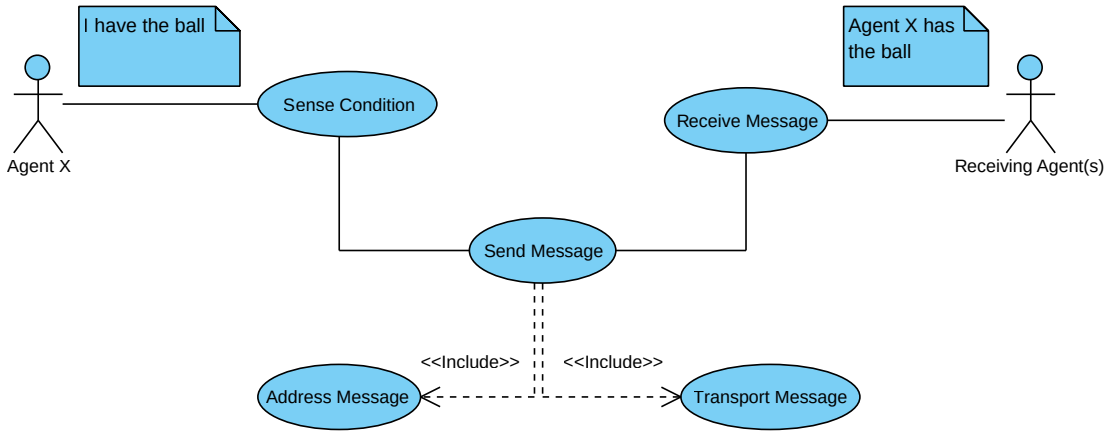


Figure 7.13: Robot Messaging Use Case.

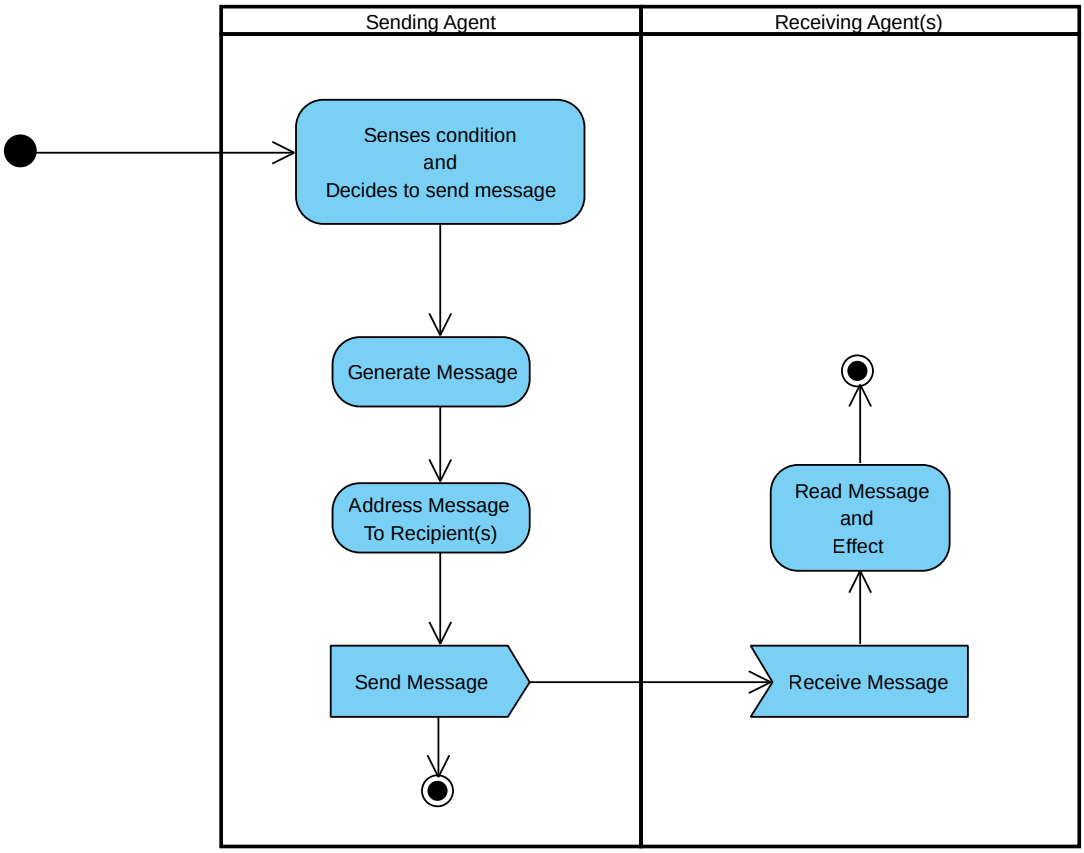


Figure 7.14: Messaging Activity Diagram.

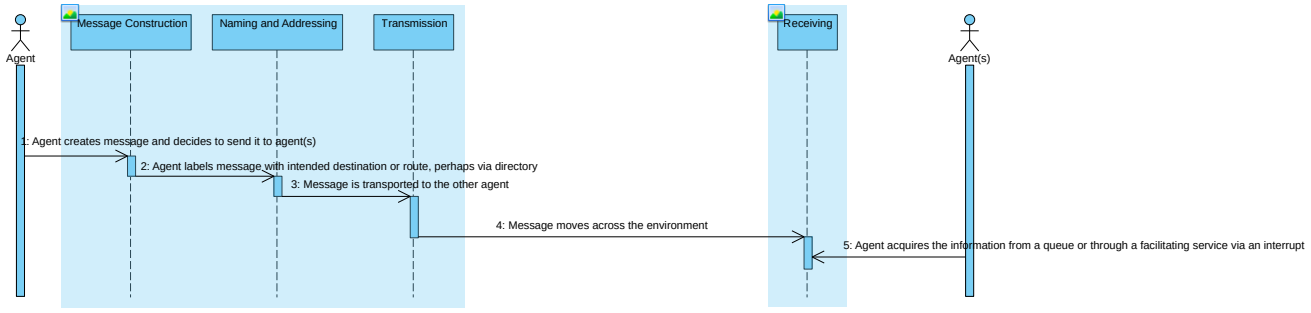


Figure 7.15: Messaging Sequence Diagram.

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

Sending Framework. The sending framework API is called for sending a message. The message body is constructed and wrapped in an envelope containing 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.

Sending Platform. The operating system and network driver receive a low-level request to build a message that is identical to the above. The message is broken up into packets that are sent individually.

Sending Host. The network card physically transmits the packets through the network pins or over the wireless antenna.

Environment. The environment in this scenario is the actual network – the Ethernet cables or Wi-Fi radio range. The packets are transmitted over the medium.

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

Receiving Host(s). The network card physically receives the packets from the network or via the wireless antenna.

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

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.

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."

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 "sees" 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 Figures 7.16, 7.17 and 7.18.

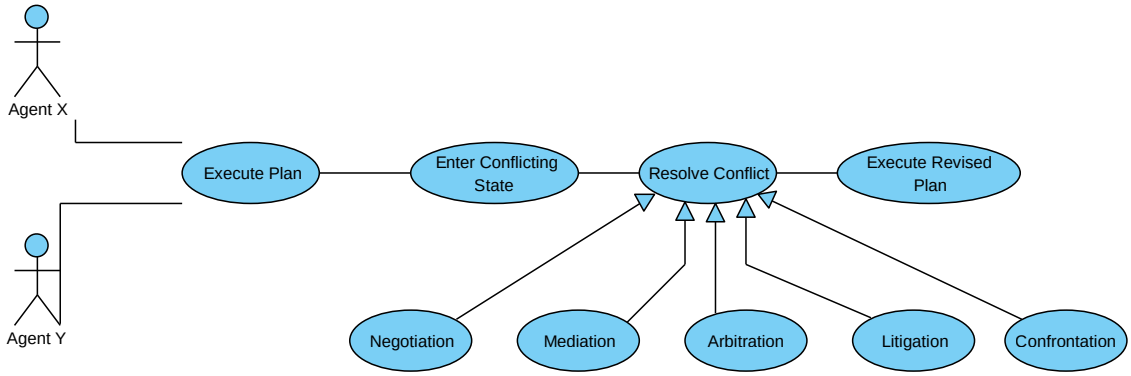


Figure 7.16: Robot Conflict Management Use Case.

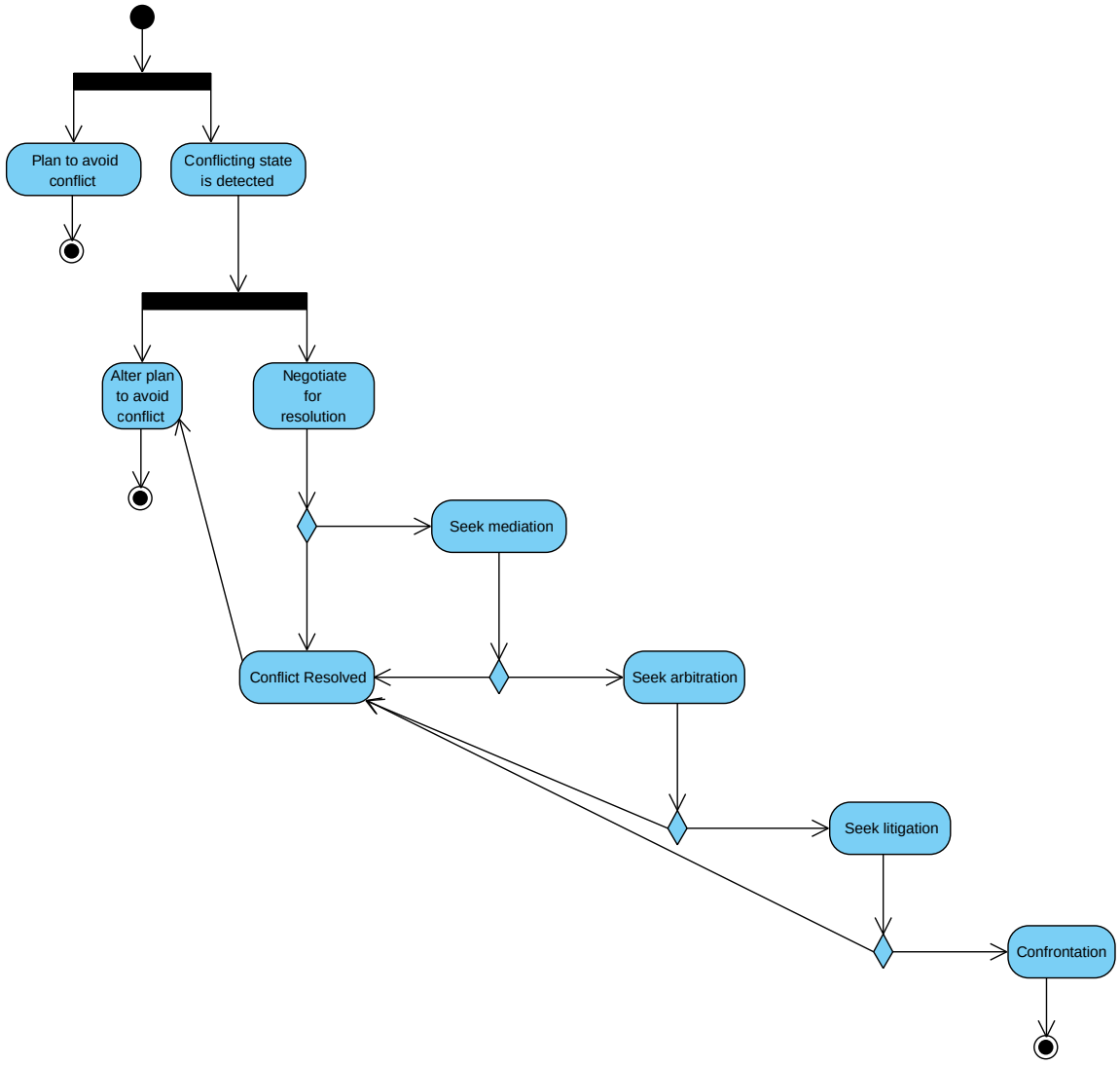


Figure 7.17: Conflict Management Activity Diagram.

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 re-plans 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.

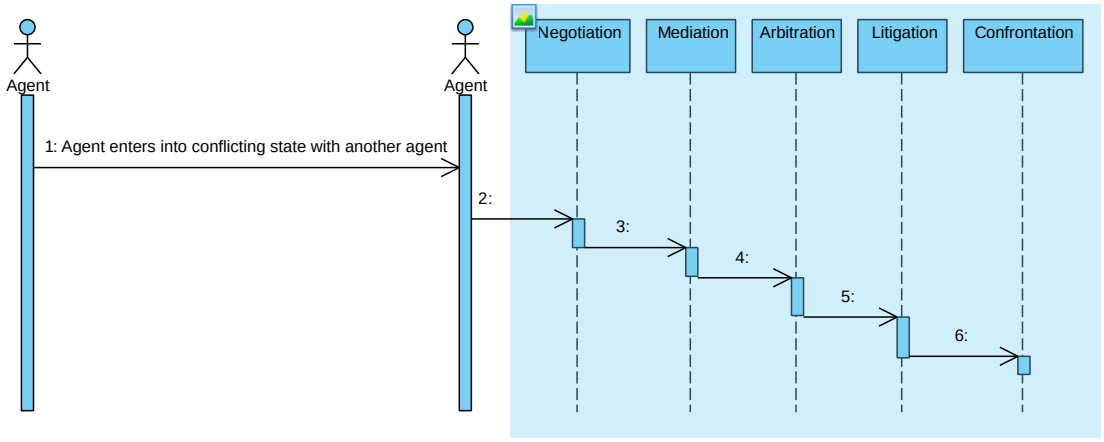


Figure 7.18: Conflict Management Sequence Diagram.

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.

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 opponent's 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.) are 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, that it wishes to communicate with a group of agents or migrate to another host. This functionality is protected through some controlling authority that 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.

Regardless, the agent authenticates itself with the controlling authority by sharing credentials such as a username and password. This information is validated and the authentication stage complete. 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 the functionality. This can be achieved in a number of ways, and is often effected through the use of encryption keys. See Figure 7.19. This process is illustrated in more detail in Figures 7.20 and 7.21.

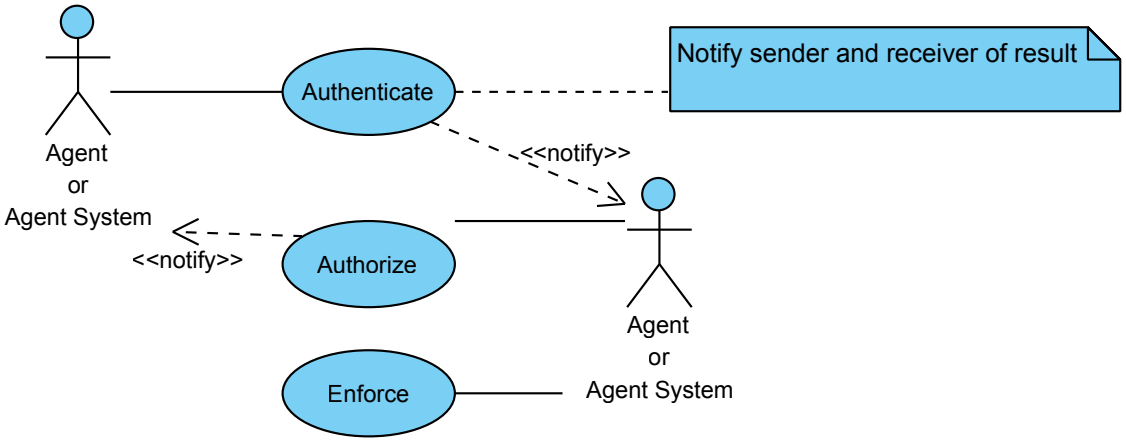


Figure 7.19: Security Use Case.

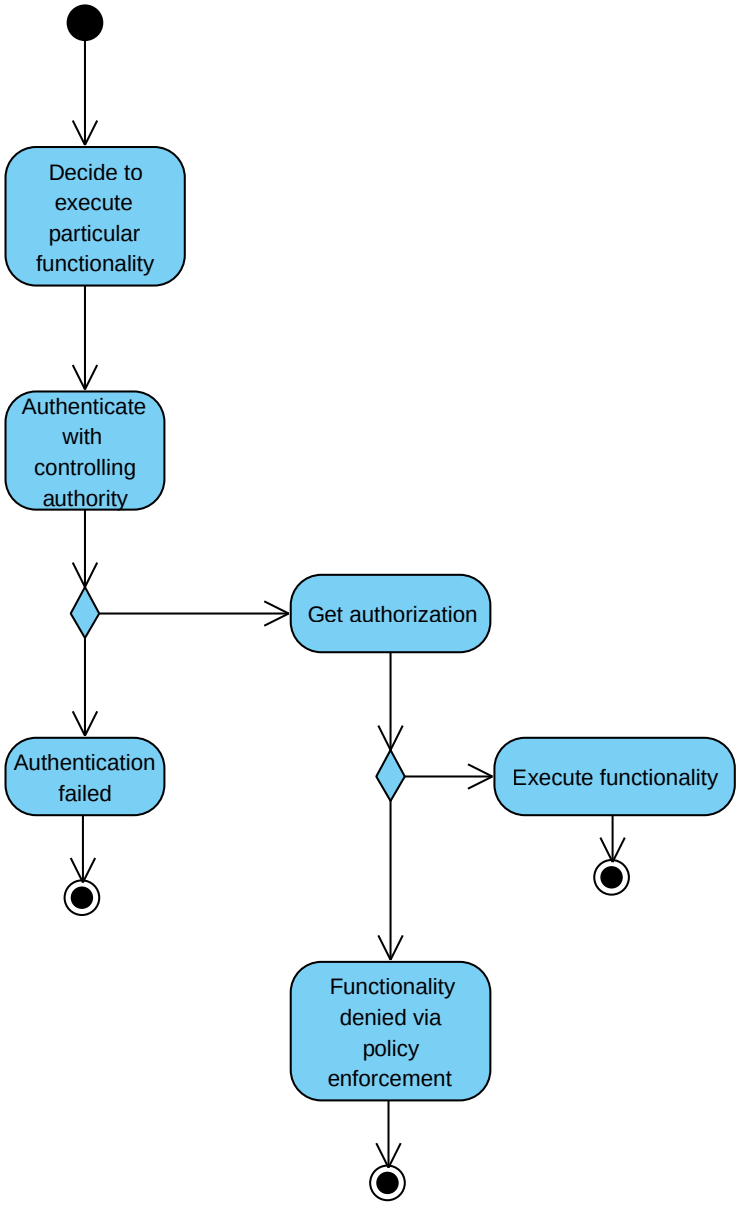


Figure 7.20: Security Activity Diagram.

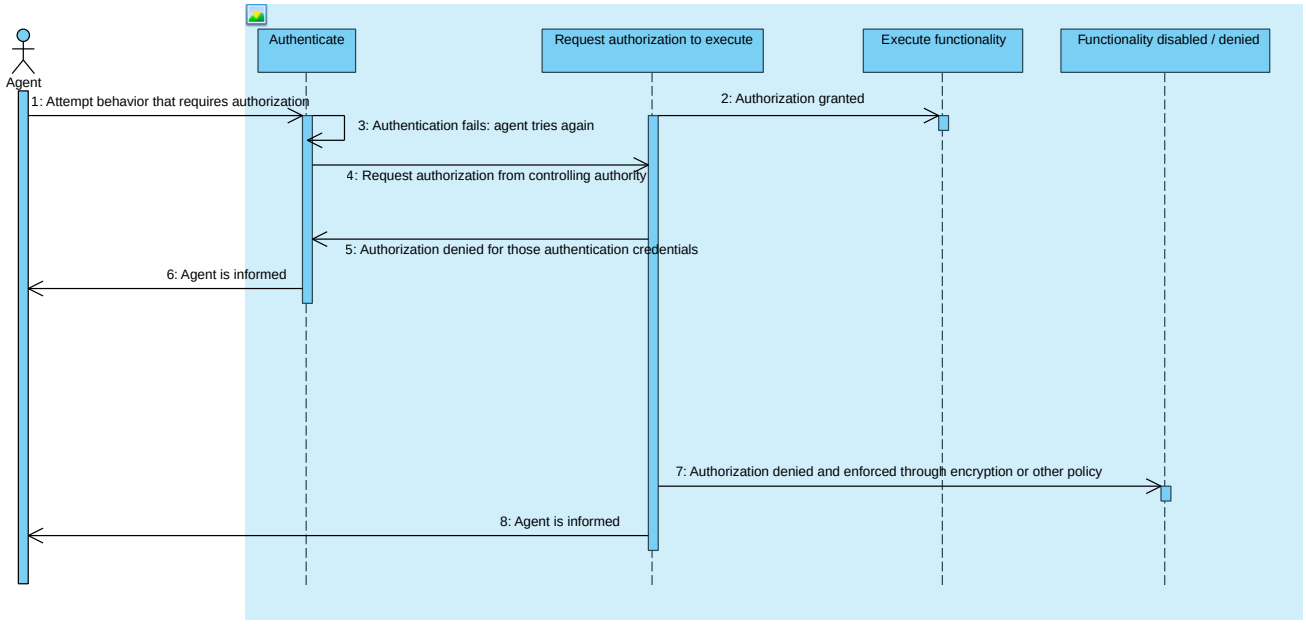


Figure 7.21: Security Sequence Diagram.

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 and then transporting to another agent framework instance, as defined by the reference model and illustrated in Figure 7.22.

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 is made to migrate, the agent is serialized by the framework, transported through the network as appropriate, and deserialized at the destination. This is illustrated in Figure 7.23 and described in detail by Figure 7.24.

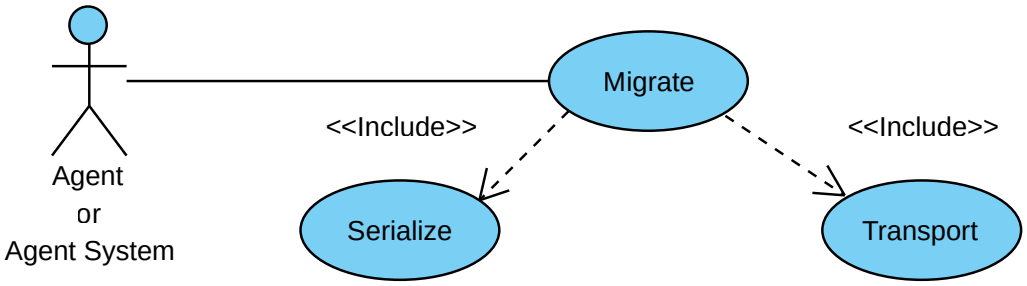


Figure 7.22: Mobility Use Case

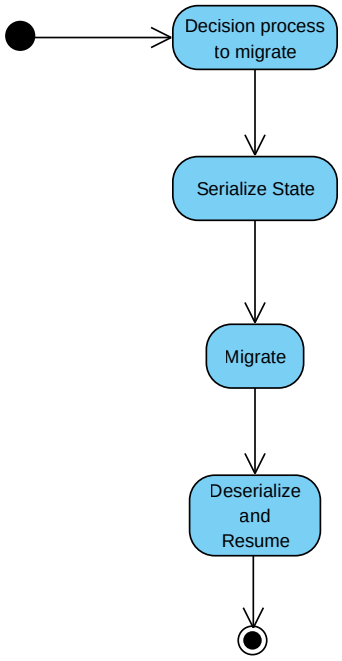


Figure 7.23: Mobility Activity Diagram

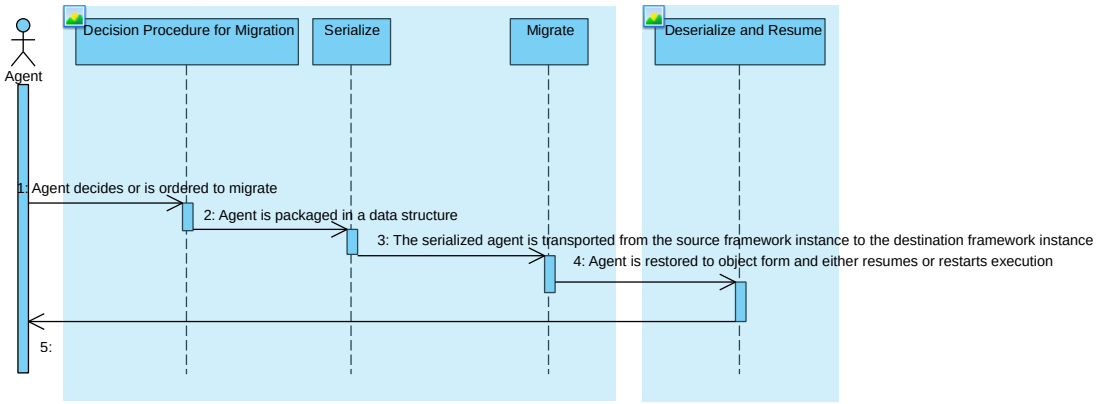


Figure 7.24: Mobility Sequence Diagram

7.2.5 Situated Agent Example: Secure Wireless Agent Testbed (SWAT)

The Secure Wireless Agent Testbed (SWAT) is a unique facility developed at Drexel University to study integration, networking, and information assurance for next-generation wireless mobile agent systems. It is the only implemented system that fully integrates:

- Mobile Agents;
- Multi-hop, Mobile Ad-hoc Wireless Networks (MANETs);
- Security and Information Assurance;

SWAT's agent-based applications are implemented in Java and were tested with the EMAA agent framework. This subsection details how it implements the core components of the ASRM.

Messaging

Messaging services are provided by the EMAA framework. These services are used to support communications among application agents providing end-user tools such as a Whiteboard, Voice over IP (VoIP), GPS tracking and other components.

Scenario. In order to accomplish the distribution of real-time GPS data, each node receiving GPS input then securely sends this information to all other hosts on the network. 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 in Figure 7.25. The messaging process is outlined in Figure 7.26.

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

Mobility

Agent mobility is handled strictly through the EMAA framework services.

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 Figure 7.27.

Security

The Security Manager is a central part of the SWAT architecture. It 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 Certificate Authority (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

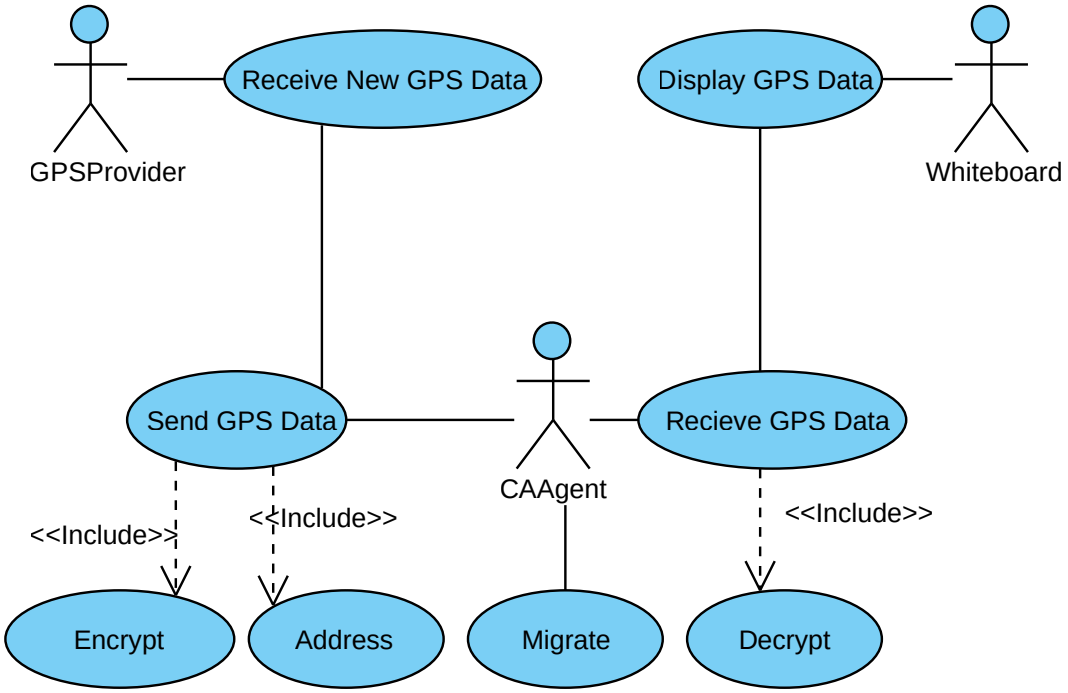


Figure 7.25: SWAT Messaging Use Case Diagram.

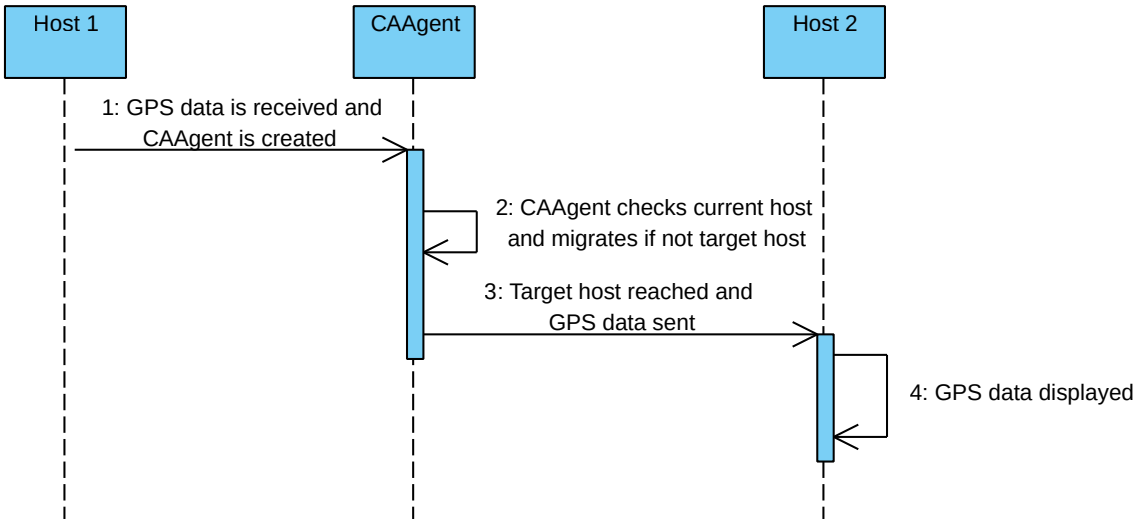


Figure 7.26: SWAT Messaging Sequence Diagram.

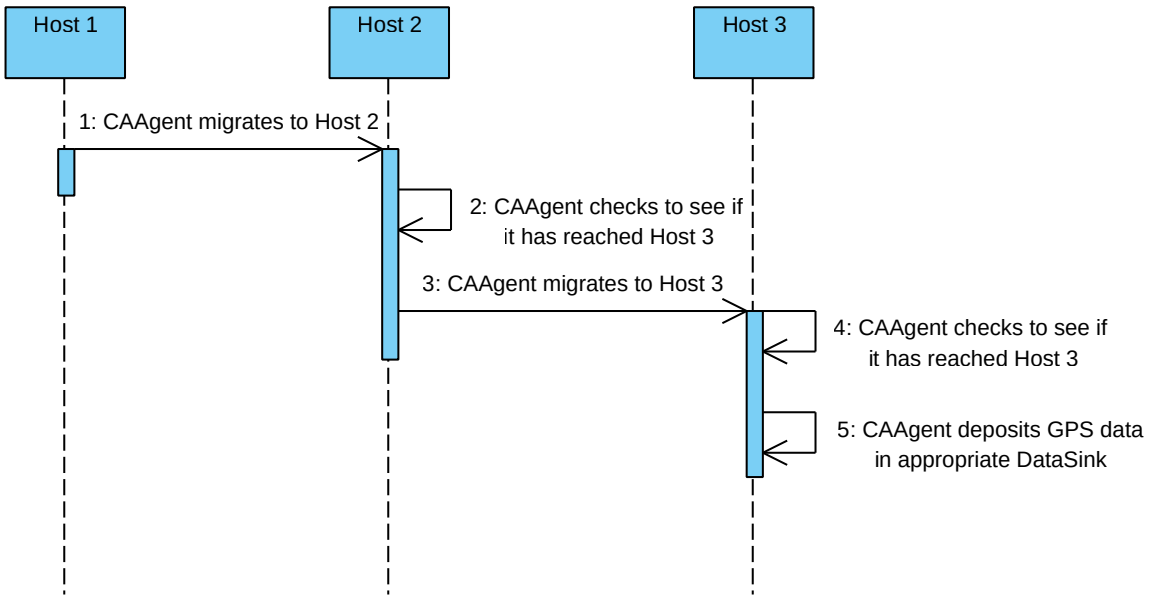


Figure 7.27: SWAT Mobility Sequence Diagram.

still be sent through revoked hosts because they can pass them along without being able to decrypt them.

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 for explicitly in the agent framework itself.

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` 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 Figure 7.28.

Group Management. SWAT makes use of the Spread toolkit for communication services for security and group management. In this context, it is a separate communications channel from the EMAA agent-to-agent messaging. Spread is a fault-tolerant messaging service. As mentioned in the Security subsection, 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 so that intra-group communication is secure. Group members can share all sorts of data: whiteboard annotations, VoIP messages, and

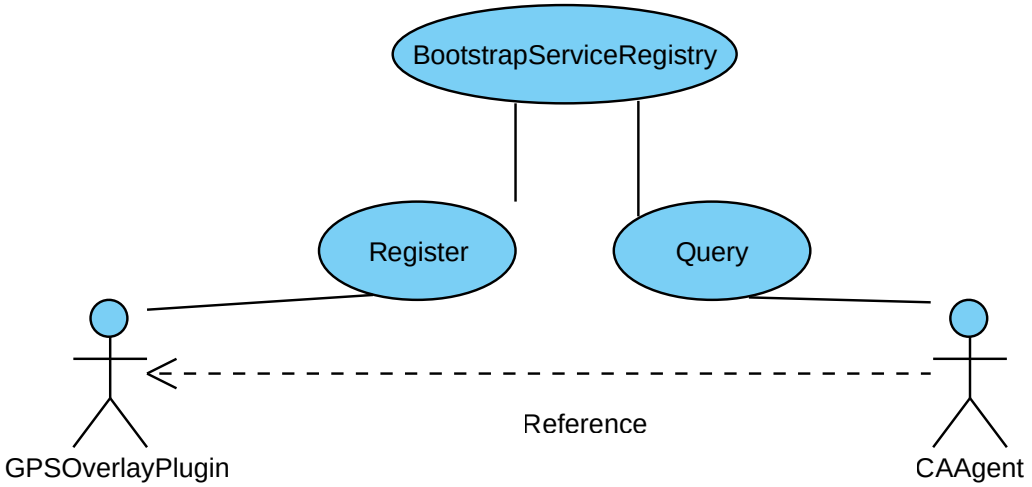


Figure 7.28: SWAT Resource Management Use Case Diagram.

GPS information. Due to security, intergroup communication is not possible except with the CA for obvious reasons. Any major event, such as a join or a leave, spawns a re-keying sequence.

Routing. SWAT has used several MANET routing protocols to create network routes at OSI layer 3. These protocols are separate from the agents, however they are necessary for agent-to-agent communication. The Optimized Link State Routing (OLSR) 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—numbers of hops—is the primary factor in determining a route; other factors, such as link quality, are secondary.

7.2.6 Example: Viruses as Agents

Computer viruses are self replicating computer programs that propagate by inserting themselves into other executable files, usually unknown to the computer's operator. In a sense, viruses may be thought of as a primitive and malicious form of an agent.

Computer worms could also be thought of as a kind of agent. In fact, the original worms, created in 1978 when computer cycles were more scarce, copied themselves to different computers in Xerox's lab looking for idle CPUs that could execute jobs[1]. Currently, the most common type of worm a computer user encounters is a malicious worm such as the *ILOVEYOU worm* (commonly known as the *ILOVEYOU virus*).

Both viruses and worms are:

- **Situated:** Situated in the infected host;
- **Proactive:** A human does not give them instructions, they spread on their own; and,

[1]The Worm Programs, Comm ACM, 25(3):172-180, 1982.

- **Interactive:** Viruses generally watch for other executing programs to find new targets, and then infect them.

Generally these viruses simply watch for other programs to execute, and copy their machine code to these files, infecting them. Once infected, a program loads the virus when it is executed, and the cycle begins again.

The functional concepts discussed by the ASRM are present in computer viruses and worms as well.

Agent Administration

This component is generally implicit in the design of the virus. Most viruses and worms will load themselves into memory when an executable program they have infected is executed. Often, they will also not start-up if an instance of themselves is already running.

Security

As defined in 4.2, security is the prevention of the execution of undesired actions. The main undesired action viruses wish to prevent is their annihilation by anti-virus scanners. Many more advanced computer viruses consisted of an encrypted form of the virus, and code for decrypting the virus. This was an attempt to prevent anti-virus programs from locating them. Some viruses also watch for anti-virus programs to start up, and attempt to circumvent their scanning mechanisms to avoid detection.

In addition, worms are often stopped by scanners on e-mail servers. To make themselves harder to detect, worms will often vary the subject and text of their messages.

Mobility

In the world of viruses, humans inadvertently facilitate the mobility of viruses. Viruses in memory watch for other programs to execute, and they then infect these programs. If these programs are copied to other computers, the virus has now infected a new host. The migration processes of viruses is the low-tech transfer of files to a portable media, which is then used in a new machine.

Worms migrate to other computers over the network. In the case of the *ILOVEYOU worm*, the worm e-mailed itself to addresses in the address book on the infected computer.

Directory Services

Viruses circumvent operating system calls, in order to find programs to infect. In a sense, this is a crude form of directory services, as it is used to locate “resources” that can then be exploited.

7.3 Example Instantiation: CoABS Grid

7.3.1 Overview of the CoABS Grid

The CoABS Grid is a Java based framework that facilitates the creation of services and agents that are discoverable and may be communicated with using either synchronous or asynchronous means. The Grid relies on the Java Remote Method Invocation (RMI) to permit methods within classes running on remote machines to be executed. Agents are based upon the basic construct of services within the Grid. An agent is an extension of the service with the addition of a message queue to permit asynchronous communication. The Grid provides registration and discovery of services (and therefore agents) via Java Jini.

It is important to note that the Grid is not a Multiple Agent System (MAS), an agent development tool, or even an agent framework per se. It is instead the group of classes, interfaces, and services necessary to support agent and service interaction. It is essentially a Service Oriented Architecture (SOA) that uses Message Oriented Middleware (MOM) to support agents. It can be thought of more as the electrical wires running throughout the infrastructure of a building than the appliances that plug into that electrical system.

The Grid does include agents, such as the mechanisms for registration and discovery, security, agent mobility, and of course agent communication. The Grid is independent however on agent capabilities and functionalities beyond these basics. There are numerous working examples demonstrating the extension of the Grid. This includes FIPA standard message compliance, interoperability with existing agent frameworks, sensor monitoring, and even control of robotic entities.

7.3.2 Mapping of the CoABS Grid

This section examines the functional areas defined by the reference model and map each to the same or similar areas within the Grid. See Figure 7.29.

Command and Control

The CoABS Grid does not have explicit administration of agents, but instead provides a framework upon which the administration can be built. Agents are instantiated as Java objects and run as services utilizing RMI and Jini. Termination of agents is handled by explicitly calling a “terminate” method within the agent. This method takes care of the necessary housekeeping of de-registering agents (see Directory below), releasing acquired resources, and gracefully stopping agents.

The Grid provides a graphical interface for management of agents via a tool called the Grid Manager. The Grid Manager permits the starting and stopping of core processes for the Grid, and monitoring agents that are registered within the Grid.

The core processes for the Grid Manager consist of a HTTP listener, a Remote Method Invocation Daemon (RMID) service, and a Jini Look-up Service (LUS). It is necessary for each of these components to be running on at least one machine on the local area network.

The Grid Manager does not provide a method for starting and stopping individual agents. However, it does allow the monitoring of agents by associating a user interface with the agent. This interface can be invoked by the GridManager or other applications to monitor and control the agent.

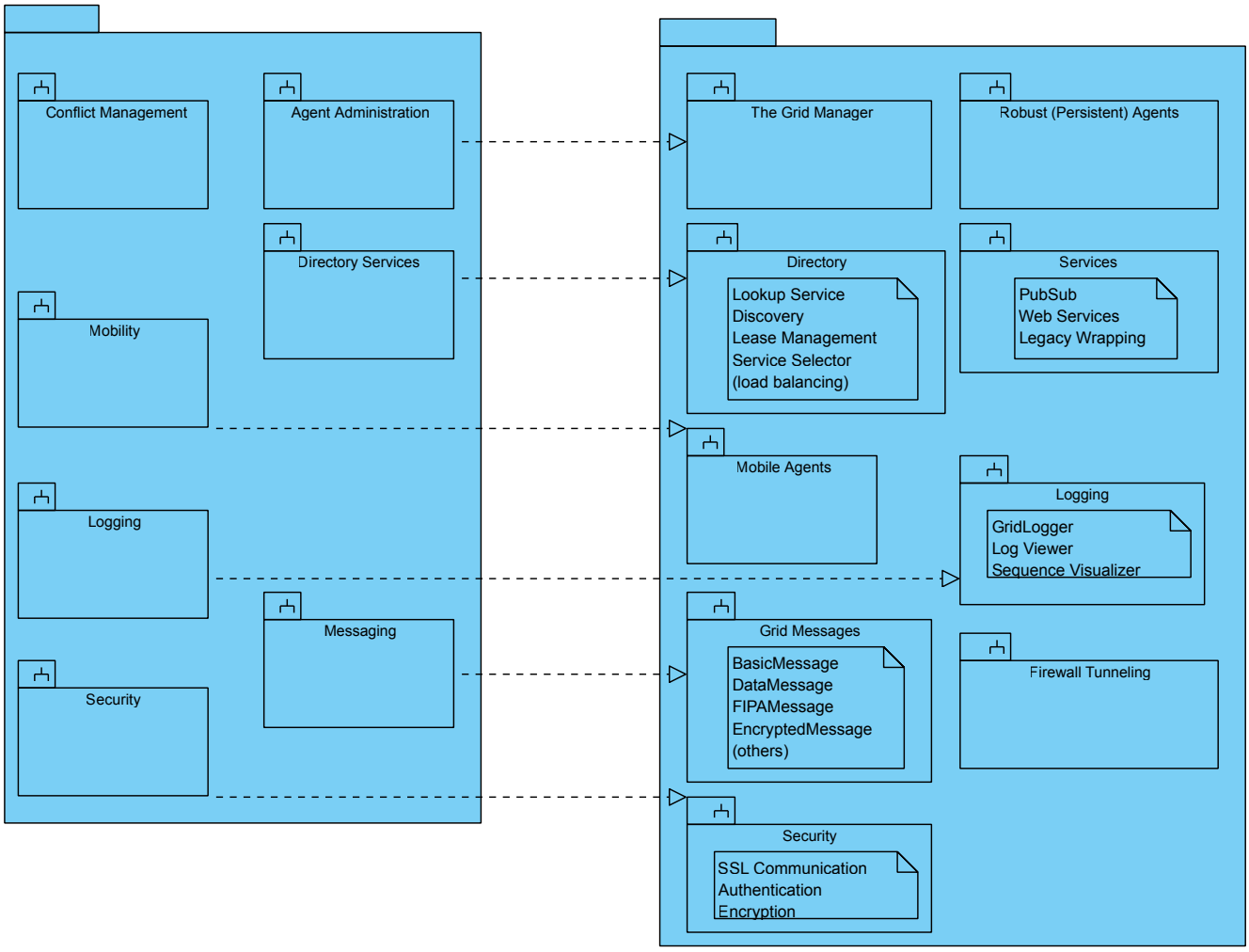


Figure 7.29: CoABS ASRM Mapping

Agent Administration

Agents under the Grid run in a Java Virtual Machine (JVM) and thus have their resources allocated and controlled via the JVM. Access to specific resources is controlled via the Java Policy implementation. The Grid has a default Policy File that is distributed as part of the standard Grid release. This Java Policy File can be modified to provide specific resource access control.

Control and access of low-level resources such as CPU's and network cards are not supported directly by Java. Instead Java has the Java Native Interface (JNI) allowing platform specific code to be written in a language such as C and interfaced to the JVM. The Grid has JNI wrappers for process control, network management, and other low level functions.

Security and Survivability

The Grid has four security aspects within its architecture. The first is the Java Policy implementation discussed in the previous section. The second is encryption of messages. Third is the digital signature of Grid agent "proxies." Finally there is support for communications via secure sockets within the Grid.

JVM Security. The lowest level of security within the Grid is provided by Java. Java security consists of several layers built directly into the JVM. This includes control of memory access, byte-code verification, and the inability to access low-level resources directly, such as the CPU and storage devices.

Policy and Permissions are used by the Grid to fine tune access by "untrusted" classes. The Grid does this by instantiating its own `SecurityManager` object and using a custom Policy File.

Message Encryption. The Grid includes the ability to encrypt and decrypt messages sent via the Grid. Encryption uses the classes in the `CoabsGrid.security` package to create keys, encrypt messages using one of the encryption standards, and decrypt using the selected standards. By default, the `GridSecurityService` uses X509 certificates, a JKS keystore, and generates signatures using the SHA1 hash algorithm in conjunction with the DSA encryption algorithm. Object encryption uses the Data Encryption Standard (DES). These tools may be utilized together or individually depending upon user's requirements. All of the capabilities use the standard Java security architecture and algorithms can easily be added without modifying the code.

Digitally Signed Agents. The Grid uses JINI and RMI by default to support discovery and remote invocation of service methods. Agents are services containing a message queue to enable communications. In order to send a message to an agent, it is necessary to have an "agent proxy" in the local JVM that communicates with the actual agent running on another platform. This proxy is downloaded from the agent's host platform.

The Grid provides a mechanism to digitally sign these agent proxies in order to ensure that the agent proxy downloaded is not modified or substituted during the communication process. This authentication is provided by methods within the CoABS Grid. These methods are `signProxy`, `verifyProxy`, `getCertificateChain`, and `checkTrust`. The

`signProxy` method accepts a service proxy object and return a `SignatureEntry`. This `SignatureEntry` can be added to the advertised capabilities of the service (see the Discovery section for more on advertisements). When clients locate the service they can use the `SignatureEntry` to verify the signature of the service and decide whether or not to use the proxy.

Secure Communications. The Grid features the option to use Secure Socket Layer (SSL) communications via RMI to provide secure communications over the network. This is accomplished in the Grid by simply using the `SecureMessageQueue` class instead of the `MessageQueue` class. This forces any clients to use SSL to communicate securely with the agent. In addition to the different message queue, the clients also need to be configured to trust the agent provider. This is done by importing the agent provider's certificate into the client's keystore.

Mobility

The Grid provides code mobility via RMI as described in the Security section. However the Grid also provides for mobility of agents. It does this by permitting one agent to send a copy of itself to another agent via a message. The sending agent locates another agent capable of receiving agents in a message. Next the sending agent serializes itself into a message and sends the message to a receiving message. Finally the sending agent de-registers itself from the directory service.

The receiving agent is a specially crafted agent whose only purpose is to receive mobile agents in messages. When it receives a message containing an agent, it re-instantiates the agent and registers it again with the directory service, communicating on a new host and socket/port as necessary.

There is also an alternative to the mobile agent technique because some users cannot permit agents to be mobile for security reasons. The Grid has a means for agents to receive their "behaviors" via messages. An agent's behavior consists of data values used to modify the way existing agent code runs. It is important to note that no code is sent to the agent. Instead the procedures .i.e. methods) already exist in the agent. The agent can change the way it runs those procedures based on the values sent via messages.

Conflict Management

The CoABS Grid provides no intrinsic ability to manage conflicts between agents. However the Grid framework is fully extensible and could be given a conflict management capability if a user desired to add it.

Messaging

The Grid provides for asynchronous messaging with agents using a message queue system. The default message queue is implemented using RMI calls to enable a client (a Java object) to place messages into an agent's queue. The agent is notified that a message was received via a listener mechanism. This basic messaging capability provides communication between agents and from non-agents to agents.

The messaging capability of the Grid is highly extensible. As discussed in the Security section, there are secure message queues and the messages themselves can be encrypted. In addition to these specialized extensions, the Grid has three basic message types.

The standard message type, `BasicMessage`, consists of message meta-information (i.e. an envelope) and a raw text field containing the actual message. The meta-information has the message identifier, performative, the time the message was sent, receiver's name, the sender's name, and optionally the sender's agent proxy. The agent receiving a `BasicMessage` typically examines the text or performative of the message to determine the type of message and decide what is to be done. This could be as simple as displaying the full message, or parsing the text of the message to pull out key fields. Agents that receive a message that includes the sender's agent proxy may reply directly to that agent without looking up the agent in the directory.

The next message type is the `AutoReplyMessage`. This message is similar to a `BasicMessage` except that the sender's agent proxy is not optional. The receiving agent uses the sending agent's proxy to send a return message acknowledging receipt, so only agent may send `AutoReplyMessages`.

The third basic message type is the `DataMessage`. This message type is the same as the `BasicMessage` except that in addition to the text payload, it contains a data object. This object can be any Java object including text, video, audio, or even the agent itself (see Mobility). The `DataMessage` is the foundation for numerous message extensions in the Grid, including the `FIPAMessage`, and XML based messages.

Finally the Grid provides a mechanism for reliable message delivery, called "Store-N-Forward" (SNF). SNF attempts to ensure message delivery by actively controlling the passing of messages. Normally sending to an agent requires an open communication channel to that agent. Under SNF, when an agent attempts to send a message to another agent and fails, the SNF mechanism automatically searches for other agents that support SNF. If such an agent is not found, the sending agent continues to try to send the message to the receiving agent or another SNF agent. This continues for a parameterized number of attempts. When an agent forwards a message to a SNF agent, the sending agent ceases its attempts to send the message to the receiving agent. It has handed off delivery responsibility to an intermediate SNF agent. The SNF agent repeatedly attempts to send the message to the receiving agent, and failing that, to another SNF agent. This continues until the message is delivered or the number of attempts specified was reached.

Logging

The Grid contains a built in logging mechanism that can be used to track messages to and from agents. Messages can be explicitly logged via methods built in to the Grid, or automatically captured by the Grid Logging service. This service is started from the Grid Manager. It listens for messages received by an agent and records them in a persistent log file. The message log file contains serialized copies of the actual messages. Therefore, all the message information, such as, message envelope (e.g. `BasicMessage`) and attached data objects (e.g. `DataMessage`), are recorded. Messages logged via the logger can be viewed with the Grid Log Viewer, an application that can be launched from the Grid Manager. The Grid Log Viewer also has a simple message replay capability.

Finally messages can be visually displayed using the Grid Agent Sequence Visualizer. This

application registers itself as a Grid logger and dynamically displays the real-time message flow between agents as a sequence diagram. This diagram is interactive and permits the user to view the message contents as well as the meta-message information.

Directory Services

The Grid relies upon the Java Jini architecture for service discovery, registration, and lease management. The key service of Jini is the Look Up Service (LUS) that is used to find services based on name, capability, or affiliation within a logical group. The Grid implementation of agents is based upon services so the Jini capabilities can be applied to Grid agents. The Grid supports three major functions within the aspect of directory support: Registration, Discovery, and Lease Management.

Registration. Agents may register themselves with the Grid directory services via Jini. An agent supplies a description of itself when it registers with the Grid. This description contains a name, a group, and a list of capabilities or advertisements. This description is used by other agents, services, and users to locate agents.

The use of advertisements is an important feature in the Grid. For example, the Store-N-Forward capability for message delivery relies upon the use of advertisements to identify agents that are able and willing to take responsibility for message delivery.

The registration functionality also includes de-registration. An agent may remove itself from the LUS without terminating itself. This permits agents to advertise themselves on the Grid for short periods of times for example. This might be used by an agent to control the number of users that it supports, perhaps based on CPU load or memory utilization.

Discovery. Users of agents (whether they be human, services, or other agents) find them via the LUS if they have registered themselves as described above. The discovery process allows look up based on filters reporting the availability of agents by name or capability. This relies on the advertisements feature described above.

Once an agent is discovered, a representative of that agent is downloaded from the Grid enabling communication with the agent. This representative is referred to as the agent proxy or "agent rep" in the Grid. The user of the agent sends messages to the agent via the proxy that in turn communicates with the actual agent.

It is important to note that once a registered agent is located via discovery, and the agent proxy downloaded, that communication with the agent no longer requires the use of the LUS. This is important because even if an agent de-registers itself from the LUS, it can still receive messages from anyone who had previously located it.

Lease Management. Given that network communication is often less then 100% reliable, it is necessary to manage the registration of agents. This is accomplished in the Grid using leases under Jini. When an agent registers itself with the Grid, it is given a lease on the registration for a certain amount of time. This time is defaulted to five minutes, but it can be changed as desired. When this lease expires, the Grid removes the registered agent from the LUS, unless the lease is renewed. Grid agents automatically renew their leases before they expire - if they are able to. So the only way an agent is deregistered, because of an expired lease, is if that agent is unable to communicate

with the Grid. This happens for various reasons, such as network failure, the agent terminates unexpectedly, or if the computer hosting the agent crashes.

Additional CoABS Grid Capabilities

Services. As mentioned previously, the Grid is based on a Service Oriented Architecture. Agents are special forms of services within the Grid. Because of this service-based design, the Grid can support additional capabilities that are not specifically attributed to agents. One example is a weather service consisting of an application that scrapes a weather web site for meteorological data and makes it available via a service. An agent in the Grid can query this service to determine the temperature and wind speed for a particular location, and then use this information as part of its environmental awareness or "belief."

Another aspect of Grid services is the ability to wrap legacy code in a service to make it available via the Grid. This permits users to integrate their existing systems with the Grid to interoperate with other systems that use the Grid.

Publisher/Subscriber. The typical model for agent communication is messaging. This messaging can be implemented in a variety of ways, but in practice it results in agents actively communicating with each other to share information.

Alternatively, information may be disseminated using as publisher/subscriber model. The Grid has a built in "PubSub" capability implemented via services. An example of a Pub/Sub application is a variation of the previously described weather service. In this case a "subscriber" locates (via the Discovery process) a service that publishes weather information. It informs that "publisher" that it is interested in receiving weather data for a particular location whenever there is a change in conditions. The publisher monitors conditions and send updates to all registered subscribers whenever there is a change.

Web Services. The Grid services were extended to support web services, including the use of UDDI and SOAP. This is not a core piece of the Grid, but rather an extension that can be utilized by those needing to interface web services with the Grid.

Autogenerator. The `AutoGenerator` is a tool that creates and compiles Java classes that wrap existing code. In addition, script files are created that can be used to re-compile and run the generated code. The tool uses reflection to examine existing Java class or jar files to extract the available methods. A user interface presents the discovered methods to the user and upon completion, Java classes are created and compiled. Using this tool, the user can easily generate a Grid agent or service without writing code.

Load Balancing. The Grid extended the basic Jini Lookup/Discovery process to add intelligent selection of services. This was primarily to support load balancing between several services or agents providing the same capability on different servers. It is important to understand that when a user searches for a service or agent using the directory function, it is entirely possible for multiple candidates to be found. Typically the LUS returns all of the matching candidates to the user and the user usually chooses one from the list.

The Grid service selector attempts to find the best service identified during the discovery process and return that one service to the requesting user. There are currently three algorithms implemented to support this, though it is easy to add more. The first is a simple random selector that attempts to achieve load balancing by distributing service requests evenly amongst the available services. The second keeps track of service usage, and returns the "Least Recently Used" service. The final implementation relies on the service to supply a "load value" to the service selector. This load value represents the percent of resource utilization, with 100% utilization meaning the service is fully utilized and is not available. The selector uses the load value to select the service with the lowest resource utilization.

Agent Persistence. The Grid utilizes the RMI activation feature to create what are referred to as "Robust Agents." These are agents that automatically restart themselves if they crash unexpectedly, for example because the system on which they are running is rebooted.

These agents are automatically started when the system is restored. Most importantly, any user of the agent is able to communicate with that agent without the need to "re-discover" the agent via the LUS. This means that while communication may be temporarily interrupted, it is restored automatically.

Firewall Tunneling. The Grid utilizes the TCP/IP protocol by default for communication. This can cause problems when the Grid is running on machines in different domains located behind firewalls. The Grid Tunnel is a service that resides behind a firewall. It communicates with other Tunnel services located on other networks that also may be behind a firewall. The Tunnel acts as a liaison to take messages and service calls and send them through the firewall using a port that the firewall permits to go through, such as port 80, and routes them to a corresponding Tunnel service delivering the information to its intended recipient.

Of course security is a major concern in network communications, and tunneling through a firewall is not done lightly. The Grid includes security features, some of which were described earlier, that ensure the Grid Tunnel services are secure. This includes communication using SSL, requiring that Grid Tunnel services can only communicate with Grid Tunnel peers, and the use of a custom Java Security Manager.

Appendix A

Agent Standards Report

A.1 Introduction

The Agent Standards Report maps the concepts of the Agent Systems Reference Model (ASRM) to both the Object Management Group's Mobile Agent Facility (OMG MAF) standard and to Foundation for Intelligent Physical Agents (FIPA) standards[1] that are suitable for instantiating them. Knowledge Interchange Format (KIF) and Knowledge Query and Manipulation Language (KQML) are also described, but only in the messaging section, since they are not meant to be a complete set of agent standards.

FIPA is an IEEE standards organization, whose goal in creating these standards is to promote interoperability between heterogeneous agent systems, as well as interoperability between agent and non-agent systems. For easy reference to the FIPA Standards all of the standards described in this document are followed by the FIPA identification number in parenthesis.

The goal of the OMG MAF standard is to promote standard interfaces in heterogeneous agent systems to facilitate interoperability. It was built on the foundation of the Common Object Request Broker Architecture (CORBA). CORBA is a framework for distributed systems that specifies an API for accessing services offered by an application, but does not prescribe how those services must be implemented.

Please note that this document does not prescribe any standards for use in agent systems. Conformance to the ASRM does not require the use of the standards described in this report.

A.2 Mapping

Figure A.1 shows a mapping of the relationship between the FIPA standards and the concepts in the ASRM. Some standards clearly map to part of a single concept, whereas other standards are more complex and describe pieces of multiple concepts. No mapping is shown for OMG MAF, because it is a single standard.

[1] Only FIPA standards classified as standard are described in this chapter.

	Agent Administration	Security	Mobility	Conflict Management	Messaging	Logging
Abstract Architecture (#SC00001)	X	X			X	
SL Content Language (#SC00008)					X	
Nomadic Application Support (#SI00014)						
Agent Management (#SC00023)	X		X		X	
Request Interaction Protocol (#SC00026)				X		
Query Interaction Protocol (#SC00027)				X		
Request When Interaction Protocol (#SC00028)				X		
Contract Net Interaction Protocol (#SC00029)				X		
Iterated Contract Net Interaction Protocol (#SC00030)				X		
Brokering Interaction Protocol (#SC00033)				X		
Recruiting Interaction Protocol (#SC00034)				X		
Subscribe Interaction Protocol (#SC00035)				X		
Propose Interaction Protocol (#SC00036)				X		
Communicative Act Library (#SC00037)					X	
ACL Message Structure (#SC00061)					X	
Agent Message Transport Service (#SC00067)					X	
ACL Message Representation in Bit-Efficient (#SC00069)					X	
ACL Message Representation in String (#SC00070)					X	
ACL Message Representation in XML (#SC00071)					X	
Agent Message Transport Protocol for IIOP (#SC00075)					X	
Agent Message Transport Protocol for HTTP (#SC00084)					X	
Agent Message Transport Envelope Representation in XML (#SC00085)					X	
Agent Message Transport Envelope Representation in Bit Efficient (#SC00088)					X	
Device Ontology (#SI00091)						
Quality of Service (#SC00094)						

Figure A.1: The FIPA ASRM Map. The rows correspond to FIPA documents, with their FIPA identifiers in parenthesis and the columns correspond to concepts in the ASRM document.

A.2.1 Agent Administration

The ASRM describes agent administration as the facilitation and enabling of supervisory command and control of agents and/or agent populations as well as allocating system resources for agents by dividing agent administration into four main processes: Agent Creation, Agent Management, Recourse Control, Agent Termination.

FIPA describes standards that relate to these processes in the Abstract Architecture Specification (#SC00001) and the Agent Management Specification (#SC00023) documents.

Agent Creation: The ASRM defines agent creation as the act of instantiating or causing the creation of agents. This encapsulates the initial allocation of resources for the agent, as well as the invocation of the agent (starting its execution).

FIPA describes agent creation as a two step process: *creation* and *invocation*. Creation is the instantiation of the agent and invocation is the starting of the agent. Upon creation of an agent, FIPA mandates that a unique identifier, AID, is given to the agent in the form of a unique name concatenated with @ and the name of the platform on which the agent resides.

In addition, FIPA prescribes that the Agent Management Service (AMS), the supervisory service on a platform, give a reference to a number of key services to the agent, in the form of a `service-root` (see the section A.2.1 for more details). The agent then optionally registers with one or more of these services, such as directory services.

The OMG MAF describes creating an agent as an interaction with the destination framework and a non-agent program, an agent from a different type of framework, or an agent of the same type of framework can create the new agent.

OMG MAF stipulates a number of steps in the process of creating an agent. First, a thread is started for agent. Next, an object of the agent's type is instantiated. Then, it is assigned a globally unique identifier (GUID), which plays a similar role as the AID in FIPA. A GUID consists of the *authority*, *identity* and *system type*. The authority is the entity creating the new agent. The identity is provided by the authority, but must be unique. The system type refers to the type of framework for which this agent was designed. Finally, the agent begins execution in its thread. Optionally, the agent registers with the MAFFinder, a directory service.

Agent Management: The ASRM defines agent management as the process by which humans or manager agents give orders to agents. FIPA does not have any standards prescribing agent management, nor does OMG MAF.

Resource Control: Resource control in the ASRM encompasses allocating system resources such as CPU and bandwidth, and providing access to resources such as user interfaces.

Services are the only method FIPA describes for accessing resources. The `service-root` — pointers to which are given to agents upon creation — provides sufficient information to contact service directory services. Agents query this resource for available services.

The OMG MAF does not prescribe any standards for resource control.

Agent Termination: Unless the MAS is meant to run forever, the agents must eventually cease execution and free its resources.

Terminating an agent in FIPA is referred to as either destroying it or quitting. If AMS forces an agent to terminate, it is called destroying, while if the AMS requests that an agent terminates or the agent initiates its own termination it is referred to as quitting. Regardless of the initiator of termination, a FIPA compliant framework requires an agent de-register with the directory facilitator and any other previously registered directory service upon termination.

In OMG MAF, an agent is terminated by calling the `agent_terminate` function with the GUID of the agent to be terminated. If the caller has appropriate permissions, the agent is termi-

nated. The entity that calls this function should also unregister the terminated agent, to prevent errors.

The system terminates an agent by providing the information in its GUID. The framework is also responsible for unregistering an agent, to prevent errors. An agent that wishes to terminate itself calls the same function, with its own GUID.

A.2.2 Security

The ASRM defines security as the preventing the 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 remain useful and dependable in the face of malice, error, or accident.

None of the FIPA standards prescribe specific security standards. However, section 11 of the FIPA Abstract Architecture Specification (#SC00001) briefly describes security features that the architects of concrete agent system must consider: Agent identity, access permissions, content validity (software code and messages), and content privacy.

The specification states that it is purposefully vague, because security issues are tightly coupled to implementation and the designer of the system must address them. An example is given in section 4.7 of using 3-DES to encrypt a message payload. Additional attributes are added to the envelope that name the encryption algorithm used and the key.

Authentication: Authentication is a process for identifying the entity requesting an action.

OMG MAF describes authentication in detail. It states that an agent must authenticate itself before performing communications with another system, or migrating to another system. Authentication cannot be done with public key cryptography, because mobile agents cannot securely carry private keys with them. Instead, the framework receiving the request or the migratory agent must authenticate the sending framework, using a method such as public key cryptography.

Authorization: Authorization is a process for deciding whether the entity should be granted permission to perform the requested action.

In OMG MAF, if the sender is authenticated using the scheme described above, then the agent is given the same authorization on the receiver that it enjoyed on the sender.

Enforcement: Enforcement is a process or mechanism for preventing the entity from executing the re- quested action if authorization is denied, or for enabling such execution if authorization is granted.

Enforcement of a security policy is not described by the OMG MAF.

A.2.3 Mobility

ASRM describes mobility as the services required to support the migration of agents between different framework instances.

FIPA's Agent Management Specification (#SC00023) describes all the mobility processes in the ASRM except for serialization. Section 2.3 of the Abstract Architecture Specification (#SC00001) describes the rational for not describing certain features, including mobility.

Decision Procedure for Migration: The ASRM states that a platform supporting mobility supports active mobility, passive mobility, or possibly both. A platform supports active mobility when the agents are responsible for deciding to migrate. However, a platform supports passive mobility when the framework is responsible for deciding when agents migrate. If the decision to migrate is made by the framework, another agent or a management service, the platform supports passive mobility.

FIPA prescribes that an agent platform support active mobility, therefore agents in a FIPA compliant system initiate the move to another host. After initiating the move, the AMS places the agent into the *transit* state. Optionally, the agent also de-registers or modifies its entry in directory services.

The OMG MAF also describes the agent as deciding to move to a remote location, although not quite as decisively as FIPA, indicating active mobility and does not mention passive mobility.

Serialize: As described in the ASRM, serialization 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.

Serialization, in this case, is the transformation of an agent into a format suitable for transmission across a communications channel. Upon transmission, the remote platform must have enough information to recreate the agent and invoke it.

Serialization is not described by a FIPA standard because it is highly dependent on the implementation details of the agent framework. For example, if the agent was written in Java or C, the implementation of serialize requires vastly different efforts. While Java only requires agents to implement the `Serializable` interface and transmit the serialized object across a network socket, a framework written in C requires a custom solution.

The OMG MAF standard describes serialization as representing the agent in a form from which the agent can be fully reconstructed. It does not describe an exact method, since this is too implementation dependent.

Migrate: Migration, which refers to the transmission of the serialized agent over a communications channel to the migration destination, is not described by a FIPA standard. The method of transmission depends on how the agent was serialized. Since serialization is not covered by a FIPA standard, migration is not covered either.

In OMG MAF, when initiating a migration, the migrating agent specifies a minimum QoS required for the migration to occur. If this is met, the agent is transmitted across the communications channel.

De-serialize, Re-Register and Resume: After the agent arrives on a host platform and is de-serialized (the specifics of which are not part of a FIPA standard), the AMS executes the agent,

at which point it resumes execution. This is analogous to invoking the agent after it is created. Optionally, the agent re-registers with the directory facilitator and other services.

When migrating, there are two options for the agent to resume its execution. The first method, called weak mobility, occurs when a migrating agent's data and code arrives on a new host and it begins execution from a fixed point. The second, called strong mobility, occurs when a migrating agent resumes execution from the point that it paused execution before migration. The OMG MAF standard explicitly states that both weak and strong mobility fall under their classification of mobility.

A.2.4 Conflict Management

Conflict Management is defined by the ASRM as managing interdependencies between agents.

FIPA has a collection of nine interaction protocols that are suitable for the conflict avoidance and conflict resolution processes. These nine protocols were not written solely to address conflict management, but can be used in such a way as to facilitate handling these situations.

The OMG MAF standard does not prescribe any standards for conflict management.

Conflict Avoidance: The ASRM defines conflict avoidance as preventing conflicts. One way agents can avoid conflict is by coordinating their individual actions to not conflict with actions taken by other agents.

To prevent conflict, it is possible to use some of the FIPA interaction protocols that allow an agent to make requests to another agent that its behavior conforms to some standard. For example, if there is a configuration file that many agents are allowed to read from and write to, care must be taken to ensure that one does not read while another is writing, as this could potentially cause inconsistent data to be read. The reader exchanges messages, conforming to the Request Interaction Protocol (#SC00026), with the agent in charge of the file, requesting that it grant the reader permission to read it. The agent in charge of the file then waits until the file is not being written to grant permission.

Conflict Detection: Conflict detection is defined as determining if a conflict is occurring or has occurred. Conflict detection is not described by any FIPA standards. In general, detecting conflicts depends on the implementation of the system.

Conflict Resolution: The ASRM defines conflict resolution as the process through which conflicts between agents' actions are resolved. The interaction protocols are ideal for negotiation in certain cases. For example, if there is a host whose CPU is being throttled, a manager agent may decide to actively manage access to the CPU so that critical applications are not delayed. This management agent sends out a call for proposal, conforming with the Iterated Contract Net Interaction Protocol (#SC00030), to all the agents that are eligible to schedule time on the CPU.

Interested agents respond with the quality of the tasks they are able to perform and the amount of CPU time required. Negotiation then continues until the management agent sends an accept proposal to the agents allowed to use the CPU.

A.2.5 Messaging

The ASRM defines messaging as the exchange of information between agents. It states that although the exchange of information between other entities, such as agents and the framework, can occur, it is not included in the messaging concept.

FIPA has many different standards involving many parts of messaging. Standards exist for encoding a message, looking up an agent to receive it, addressing it and transmitting it. Furthermore, standards exist for doing these actions in different ways, depending on the environment in which the agents are situated.

Messaging is not addressed by the OMG MAF specification. However, the specification suggests using CORBA's object communication standards for agent communication.

Message Construction: The ASRM defines message construction as the process through which a message is created, once the message has been decided upon. FIPA messages, described be the ACL Message Structure Standard (#SC00061) consist of a performative representing the purpose of the message, the message content (ie: the actual message), and meta-information such as the sender and receiver. Additionally, optional conversational control parameters can be included, such as the interaction protocol being used (if any) and an identifying name for the conversation.

The Communicative Library Specification (#SC00037) lists types of messages that can be constructed, which is to say it is a list of valid performatives. Ideally all of the communicative acts specified in this standard are supported, although, to comply, an agent is only required to implement the `not-understood` message.

The SL Content Language Specification (#SC00008) describes syntax for the content portion of the ACL message. This is the actual message; the information being communicated.

After the message is constructed from these components, a representation is chosen. Three standards, ACL Message Representation in Bit-Efficient Specification (#SC00069), ACL Message Representation in XML Specification (#SC00071), and ACL Message Representation in String Specification (#SC00070) describe how to represent a message in a specific format. Different representations are used in different environments. For example, in a bandwidth constrained environment, a bit-efficient representation makes more sense than an XML representation.

Another way of constructing messages is to use the Knowledge Query and Manipulation Language (KQML), a standard that came out of the DARPA Knowledge Sharing Effort (KSE). Like the FIPA ACL Message Structure Standard, KQML has a number of parameters such as *sender* and *receiver*, as well as a performative and some content (ie: the actual message). Another standard that came out of KSE, Knowledge Interchange Format (KIF) can be used to encode the content of KQML messages.

Naming and Addressing: The ASRM describes the naming and addressing process as the mechanism for labeling a message with its intended destination or route. Services that facilitate this labeling, such as directory white page services (to search for agents to send messages to), are considered part of this process.

A directory service such as the directory facilitator (DF) described in the Agent Management Specification (#SC00023) is the standard method of searching for agents. The

`agent-directory-service` (ADS) described in the Abstract Architecture Standard Specification (#SC00001) is slightly different from the DF, but performs a similar task. Please refer to the primary sources for more information.

The message constructed in the previous processes is then attached to an envelope, as described in the Agent Message Transport Service Specification (#SC00067). The envelope contains the information that is required to deliver the message successfully, which is the sender, the receiver, a timestamp, and a description of the acl-representation used to encode the message. (The latter is so that the recipient properly decodes the message.) Optional envelope parameters can also be included, as well as custom envelope parameters. The latter parameters must have "X-" prepended to their names.

Transmission: The reference model defines transmission as the actual transport of the message over a communications channel. There are two ways of transmitting messages described by FIPA standards in the Agent Message Transport Service Specification (#SC00067), both of which use an agent communications channel (ACC). An ACC is an abstract entity that transmits messages between different physical hosts across a communications link. In the first method, the sending agent gives its message to its local ACC, who transmits it to the ACC on the remote host where the recipient agent is located. In second method, the sending agent sends its message directly to the ACC on the remote host.

When using the first method of transmission, ACCs on different host send messages using a message transmission protocol (MTP). There are two standards describing MTPs, the Agent Message Transport Protocol for IIOP Specification (#SC00075) and the Agent Message Transport Protocol for HTTP Specification (#SC00084). An ACC, when sending a message, queries the remote ACC to determine which protocol to use.

Another way of of transmitting messages, which the FIPA standards specifically do not exclude, is agents sending messages directly to other agents. Although this method is not prohibited, it is also not described by the standards and the decision to use it is deferred to the implementers.

Receiving: Assuming one of the two methods described in the standard was used to send a message, it is delivered to the recipient by the ACC local to it. FIPA does not specify how this is done (e.g.: inserted into message queue, etc). It does specify that both the envelope and the message are delivered to the recipient, which is to say that the recipient has full access to the envelope of all messages sent to it.

A.2.6 Logging

Logging is described by the ASRM as the enabling of information about events that occur during agent system execution to be retained for subsequent inspection. Neither the OMG MAF standard nor any of the FIPA standards describe logging.

Appendix B

Survey of Surveys

B.1 Introduction

This reference model is a reflection of a wide variety of research. Information was gathered from many sources including: journals, magazine articles, conference proceedings, textbooks, white papers, and manuals for agent frameworks. In addition to these static sources, experts in the agent field provided input.

Due to the breadth of subjects classified under "agent research," classifying survey papers is not trivial; however, a few works stand out that effectively summarize the history—past, present, and future—of agents, analyze the benefits or lack thereof for using agents, or outline the way agent research should be conducted. These are regarded as surveys and usually give a brief overview of other works that exist in some category. They are helpful resources because they create a single location where a person researching a certain subject can find more references. What happens when numerous areas of research need to be considered? The creation of the reference model presented such a problem. Our solution was to create a survey of surveys that alleviates some of the overhead involved in finding works in a wide variety of fields.

This document sorts surveys that categorize works in a particular field. It is organized by subject where the actual categories are purposely broad and the scope of each is the entire agent world. This is necessary because survey papers tend to make some sort of assertion about a field. For example, a survey paper that just categorizes papers on Agent Security is not very meaningful. A better survey paper would be "Current Trends in Agent Security." All survey papers presented do comment on a field as whole and they are categorized based on how they comment on that field.

An objective of this survey is **not** to make a universal observation about the agent field or any particular subset. In the design of the reference model, the entire field was taken into account and made no attempt to draw conclusions about an individual one. The applicability of this survey is twofold. First, and most obviously, it is possible to obtain references that describe a certain field in a certain way. A person writing a survey paper often needs sources describing a specific field. A quick look into some (or possibly all) categories provides a detailed overview. A second use of this survey allows greater generalizations to be drawn regarding agents as a whole. By seeing how agent research has progressed, trends are identified and, most importantly, the amount of research in a particular field is evident. This information then allows for the comparison of fields.

B.2 Categories

Survey papers are categorized by the way in which they comment on a particular field. In such a way, the categories span all fields at once. The entirety of the categories summarizes agent research as a whole. Comparing two different categories does not make much sense since the categories are all quite different from each other. Two works within one category can be compared in a variety of ways. For example, if two works are related to different fields, it is possible to address similarities in their histories.

Following is a list of categories with brief definitions. It is important to note that the categories are not mutually exclusive; however they do not overlap. That is, their definitions are specific and exclude other categories, but surveys are often more general and comment on multiple aspects of a field so are included in multiple categories.

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.

Research Methodology These surveys explain how to conduct agent research and how the results can be applied to the real world.

Status of Research Surveys that fall into one of three sub-categories can be found here: historical overview, present trends, or predictions for the future.

Textbooks Textbooks in the typical context. They may serve a variety of purposes so they are included in their own category.

B.3 Results

The following chart summarizes each paper by its primary and possibly its secondary categorization. In some cases, papers are very specific to a category and thus only listed once in normal typeface. In other cases, there is a primary categorization, but an overlap with another category. These papers are listed twice: first in standard typeface for the primary category and in italics for the secondary category.

CATEGORY	WORKS
Analysis of Paradigm	Chess (1995) Etzioni (1995) Ghezzi (1997) Fugetta (1998) Huhns (1998) Gray (2000) Wooldridge (2000) Kotz (2002)
Research Methodology	Hanks (1993) Fonseca (2002) *Fugetta (1998)* *Jennings (1998)*
Status	Nwana (1996) Hagen (1998) Jennings (1998) Sycara (1998) Milojicic (2000) *Huhns (2000)* *Fonseca (2002)*
Textbooks	Russell (2003) Vigna (1999) Weiss (2001) D'Inverno (2001) Wooldridge (2002)

B.4 Summaries

This section provides the categorized works in a greater detail. Included is a brief summary of each work. This further directs the readers as to the appropriateness of a certain work, especially when searching for a paper from a certain field.

B.4.1 Analysis of the Agent Paradigm

Chess (1995) [5]. Chess details the life-cycle of an agent and how it migrates internally through the API and externally through the network between nodes. Benefits of using agents for certain tasks, such as supporting mobile clients, interacting between client and server, and removing a consistent state requirement are assessed with particular attention paid to viewing agents from an economic perspective. Chess concludes that mobile agents are of greatest advantage in large networks or smaller networks where communication is uncertain. In such environments, agents reduce the need to preserve state between client and server, eliminate the need for multiple communications/transactions thereby reducing bandwidth, and enable a large amount of scalability. Agent's largest disadvantage is the need for heightened security. Servers must be able to authenticate agents, authorize them to perform tasks using system resources, and detect malicious agents.

Etzioni (1995) [8]. Etzioni attempts 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.. The SoftBot is similar to a hotel concierge and must be able to notify, enforce constraints, and locate and manipulate objects. Goal specifications are defined, the architecture is mapped out, and various other properties are examined in depth. The definition of an agent is never settled upon, but the SoftBot example exemplifies a typical agent and subjects future creators to ponder some of the challenges facing agent implementations.

Ghezzi (1997) [39]. Ghezzi shows benefits of using agent technologies by comparing and contrasting separate implementations. In particular, he highlights the differences by comparing some implementations with the client-server model and a remove evaluation design. The common task is to design a Database Management System (DBMS) with a search engine. This database is assumed distributed across a network. The three categories used to compare the implementations are message-based transmissions, strong mobility, and weak mobility. No solid conclusion is formulated in the paper and the author allows the reader to decide which implementation is most advantageous.

Fugetta (1998) [37]. In this work, Fugetta explains mobile code and its applicability to the sciences. At the time of publication, not much work was done in the area of mobile code. The authors present a framework for understanding such mobile code and give a wide range of example implementations. The various design paradigms for the implementations are inspected which include client-server, code-on-demand, and the agent model among others. A main goal of the paper is to provide a guideline for developers that is to be used when deciding whether or not to use mobile code in an implementation. Fugetta concludes with a case study involving the development of network management software. The process is examined with the classifications previously mentioned being highlighted.

Huhns (1998) [46]. Huhns and Singh, who are among the leading researching in Distributed Artificial Intelligence and Multi-Agent Systems, put together 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." Aside from the documents, the authors give a good overview of agent-related terms: agents, systems, frameworks, environments, and autonomity.

Gray (2000) [40]. Gray establishes the major advantages for using agents. He admits that for every program implemented using agents, there is a better alternative; however, agents provide enough generalization to be decent solutions to a wide range of problems—and this is their primary benefit. All too often, this sacrifice is worthwhile. 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. This flexibility is

what makes agent programming so appealing and useful. Two simple examples, an information acquiring agent using distributed databases and a military operation that requires software that is not pre-installed on mobile computing devices, illustrate the benefits of using agents versus other methods. No other implementation combines the six attributes quite like agents.

Wooldridge (2000) [70]. Wooldridge 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 LORA, or the Logic of Rational Agents. This framework is then use to highlight teamwork, communication, and cooperative problem solving with regard to rational agents.

Kotz (2002) [49]. Kotz analyzes the many barriers that agent research must overcome before it can be differentiated from mobile code. The existing difference is slight. An agent requires a large architecture and framework in which to operate, while mobile code simply takes advantage of schemes already included in the architecture (such as RPCs). When using agents, programmers are forced to tailor their code to a specific framework. If standards were established, such as a reference model, then the agent code could be more generic and universal. In that regard, the focus should be on extracting the major components from the "monolithic systems" and creating an abstract representation that generalizes most agent frameworks. An approach like such would significantly reduce the overhead required to program agents. The reference model allows the agent field to establish itself as a major part of AI.

B.4.2 Research Methodology

Hanks (1993) [43]. Hanks describes common problems stemming from the way research is conducted using AI planning-oriented implementations of agent testbeds. In particular, Hanks promotes that researchers need to realize that they are working in a controlled environment and experiments merely produce comparisons between environment parameters and how the agent reacts; benchmarks and testbeds are empirical tools so should be used with caution in the theoretical realm. Results obtained in this manner do not comment on theory; instead these results should be used to compare performances.

Fonseca (2002) [10]. Fonseca recognizes 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 believes this approach is awry. 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.

B.4.3 Status of Agent Research

Nwana (1996) [58]. Nwana explores the definition of an agent. She presents a Ven Diagram consisting of cooperation, learning, and autonomous in order to define an agent. Within each of

these possible classifications, there are sub-classifications 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 some challenges facing implementation. These qualities give rise to the future direction of agent research.

Hagen (1998) [42]. Hagen discusses the impact of agents on various mobile object middlewares, such as CORBA. Further, the article describes how certain mobile agent platforms are applied to different environments—focusing on telecommunications. Mobile agent solutions are being developed for a handful of fields within telecommunications: business, network, virtual home, and personal communication. The next step is to formalize this agent work and build solid platforms that these agents can use to conduct their business. Agents have a great potential in the telecommunications world and, if standardized and implemented correctly, will be of great benefit.

Jennings (1998) [48]. Jennings paper attempts to provide "order and coherence" to the field of agent technologies. He begins with a brief overview of agents, highlighting his 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 concludes by noting that agent-based research is a new field with great potential that will find a variety of applications upon its maturity.

Sycara (1998) [65]. Based on a definition of MASS that involve limited viewpoints, no global control, decentralized data, and asynchronous computation, Sycara puts forth several problems that currently face and will face developers of such agent frameworks. At a lower level, agents in MASs must contain the ability to reason, organize themselves, and share the workload in order to operate efficiently. These issues are addressed using several types of planning and modeling to enhance effectiveness. There exist many different MAS that have been already developed including, but not limited to, RETSINA. Sycara concludes by examining RETSINA and highlighting the aforementioned lower-level properties.

Milojicic (2000) [57]. Milojicic's book is a survey on mobility with a domain that extends to not only agents, but processes and computers as well. It presents a distinguished set of papers from years of research. The survey begins with a collection of papers on mobility. It includes process migration, user-space migration, and an assortment of migration policies. The survey then shifts gears and focuses on mobile computing—for example, using laptops. Finally, the authors switch gears to mobile agents and their applicability to communications, such as the Internet. Overall, the book proves that these three areas are quite similar and hints at the future directions for each.

B.4.4 Textbooks

Vigna (1999) [68]. This is work is part of the *Lecture Notes in Computer Science* series. It is intended to be an overview of MASs with a focus on the security aspects. The author begins by differentiated agents from remote procedure calls and describes applications where both are advantageous. The rest of the book is a collection of papers that deals with security aspects of agency. Each paper is written by different authors who are experts in their respective fields: David Chess, Giovanni Vigna, James Riordan, to name a few. Different security mechanisms are first analyzed and compared. Then, actual implementations are inspected for the security measures they provide.

D'Inverno (2001) [7]. This textbook is a standard introduction to agent technologies. It gives an in depth overview of agents and agent frameworks by examining the SMART agent framework. This agent framework is used to describe various aspects of agency, such as relationships, interactions, and types of agents. The definitions presented are quite formal and there exists a plethora of mathematical notation throughout the book. The book concludes with an application of the SMART framework entitled actSMART.

Weiss (2001) [69]. Weiss' main objective for creating this book was to provide a textbook for a field that lacked them. Hence, he created an MAS textbook that focuses on Distributed Artificial Intelligence. Each chapter is written by different leaders in the agent field. Authors include: Michael Wooldridge, Michael Huhns, Edmund Durfee, and Munindar Singh among others. A general definition in the first chapter leads to the discussion concerning agent systems and societies of agents. The author then focuses on societies of agents and describes types of interactions. For example, distributed problem solving and planning is examined as well as learning and logic-based reasoning. This textbook is not as formal in regard to its definitions, except where it needs to be (i.e. the Logic-Based Representation and Reasoning chapter). It does offer an algorithmic approach to many common agent procedures by providing pseudo-code.

Wooldridge (2002) [71]. Wooldridge wrote this textbook as an introductory text to agents and multi-agent systems. A brief history is followed by an exposition concerning individual agents. The usual, broad definition of agents is given and then reasoning and reacting is 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. Thus, this textbook gives a good overview of agents, agent systems, and applications of agent systems.

Russell (2003) [61]. The Russell & Norvig textbook on Artificial Intelligence is one of the most popular textbooks for AI courses. It is not specific to agents, but it takes an agent approach to describing the basics. Topics that are covered include problem solving, knowledge and reasoning, and planning. The book progresses into the more advanced topics of reasoning and learning. The textbook concludes with an in-depth discussion of agents and their applications in the real world as robots.

B.5 Conclusion

This paper looks at works from various fields in the agent community: software agents, intelligent agent design, multiagent systems, and DAI to name a few. Though interrelated, each subject area has a well-defined research track. Instead of focusing on a particular subset of agents research, the agent community in its entirety is considered and a practical resource to aid in finding works that describe a field in a particular way or finding works to use in comparing different fields is provided.

The evolution of agent research, as a whole, is made quite obvious by these documents. For instance, an early trend was to define agents [58, 8]. This proved to be near impossible and current works tend to accept and work with the abstract definition that includes sensing, reasoning, and acting. Security was mentioned as a major problem that agent research would face in Chess's work [5]. There now exist several works, such as [68], that show how agent security has become a primary focus in all aspects of the agent world. Then there are the problems that remain unsolved, such as how to precisely define intelligent and how to make agents that exhibit such characteristics [70].

Agent research has come a long way in the past decade. It began, as most sciences do, with discussions and arguments over what agents are and how they are applicable. Soon afterwards, theory was developed with empirical evidence to back it up. Now, agent research thrives at the forefront of artificial intelligence—at the forefront of technology in the new millennium.

Bibliography

[1] Dublin core metadata initiative, November 2006. http://dublincore.org/.

[2] Tony Andrews, Farncisco Curbera, Hitesh Dholakia, Yaron Goland, Johannes Klein, Frank Leymann, Kevin Liu, Dieter Roller, Doug Smith, Satish Thatte, Ivana Trickovic, and Sanjiva Weerawarana. Business process execution language for web services, May 2003.

[3] David Beckett. Rdf/xml syntax specification (revised), February 2004.

[4] Tim Bray, Jean Paoli, C.M. Sperberg-McQueen, Eve Maler, and Francois Yergeau. Extensible markup language (xml) 1.0 (third edition), February 2004.

[5] David Chess, Colin Harrison, and Aaron Kershenbaum. Mobile agents: Are they a good idea? Technical Report RC-19887, IBM, Yorktown Heights, NY, December 1994.

[6] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web services description language (wsdl) 1.1, March 2001.

[7] Mark D'Inverno. *Understanding Agent Systems*. Springer-Verlag, New York, NY, 2001.

[8] O. Etzioni and D. S. Weld. Intelligent Agents on the Internet: Fact, Fiction, and Forecast. *IEEE Expert*, 10(4):44–49, August 1995.

[9] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L.Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – http/1.1. RFC 2616 (Best Current Practice), June 1999.

[10] Steven P. Fonseca, Martin L. Griss, and Reed Letsinger. Agent behavior architectures a MAS framework comparison. In Maria Gini, Toru Ishida, Cristiano Castelfranchi, and W. Lewis Johnson, editors, *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'02)*, pages 86–87. ACM Press, July 2002.

[11] Foundation for Intelligent Physical Agents. Abstract architecture, December 2002. http://www.fipa.org/specs/fipa00001/.

[12] Foundation for Intelligent Physical Agents. Acl message representation in bit-efficient encoding, December 2002. http://www.fipa.org/specs/fipa00069/.

[13] Foundation for Intelligent Physical Agents. Acl message structure specification, December 2002. http://www.fipa.org/specs/fipa00061/.

[14] Foundation for Intelligent Physical Agents. Agent message transport protocol for http, December 2002. http://www.fipa.org/specs/fipa00084/.

[15] Foundation for Intelligent Physical Agents. Brokering interaction protocol specification, December 2002. http://www.fipa.org/specs/fipa00033/.

[16] Foundation for Intelligent Physical Agents. Contract net interaction protocol specification, December 2002. http://www.fipa.org/specs/fipa00029/.

[17] Foundation for Intelligent Physical Agents. Device ontology specification, December 2002. http://www.fipa.org/specs/fipa00094/.

[18] Foundation for Intelligent Physical Agents. Fipa acl message representation in string specification, December 2002. http://www.fipa.org/specs/fipa00070/.

[19] Foundation for Intelligent Physical Agents. Fipa acl message representation in xml specification, December 2002. http://www.fipa.org/specs/fipa00071/.

[20] Foundation for Intelligent Physical Agents. Fipa agent management specification, March 2002. http://www.fipa.org/specs/fipa00023/.

[21] Foundation for Intelligent Physical Agents. Fipa agent message transport envelope representation in bit efficient specification, December 2002. http://www.fipa.org/specs/fipa00088/.

[22] Foundation for Intelligent Physical Agents. Fipa agent message transport envelope representation in xml specification, December 2002. http://www.fipa.org/specs/fipa00085/.

[23] Foundation for Intelligent Physical Agents. Fipa agent message transport protocol for iiop specification, December 2002. http://www.fipa.org/specs/fipa00075/.

[24] Foundation for Intelligent Physical Agents. Fipa communicative act library specification, December 2002. http://www.fipa.org/specs/fipa00037/.

[25] Foundation for Intelligent Physical Agents. Fipa device ontology specification, December 2002. http://www.fipa.org/specs/fipa00091/.

[26] Foundation for Intelligent Physical Agents. Fipa query interaction protocol specification, December 2002. http://www.fipa.org/specs/fipa00027/.

[27] Foundation for Intelligent Physical Agents. Fipa sl content language specification, December 2002. http://www.fipa.org/specs/fipa00008/.

[28] Foundation for Intelligent Physical Agents. Iterated contract net interaction protocol specification, December 2002. http://www.fipa.org/specs/fipa00030/.

[29] Foundation for Intelligent Physical Agents. Message transport service specification, December 2002. http://www.fipa.org/specs/fipa00067/.

[30] Foundation for Intelligent Physical Agents. Nomadic application support specification, December 2002. http://www.fipa.org/specs/fipa00014/.

[31] Foundation for Intelligent Physical Agents. Propose interaction protocol specification, December 2002. http://www.fipa.org/specs/fipa00036/.

[32] Foundation for Intelligent Physical Agents. Recruiting interaction protocol specification, December 2002. http://www.fipa.org/specs/fipa00034/.

[33] Foundation for Intelligent Physical Agents. Request interaction protocol specification, December 2002. http://www.fipa.org/specs/fipa00026/.

[34] Foundation for Intelligent Physical Agents. Request when interaction protocol specification, December 2002. http://www.fipa.org/specs/fipa00028/.

[35] Foundation for Intelligent Physical Agents. Subscribe interaction protocol specification, December 2002. http://www.fipa.org/specs/fipa00035/.

[36] Martin Fowler and Kendall Scott. *UML distilled (2nd ed.): a brief guide to the standard object modeling language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[37] Alfonso Fuggetta, Gian Pietro Picco, and Giovanni Vigna. Understanding Code Mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, May 1998.

[38] M. R. Genesereth and R. E. Fikes. Knowledge Interchange Format, Version 3.0 Reference Manual. Technical Report Logic-92-1, Computer Science Department, Stanford University, Stanford, CA, USA, June 1992.

[39] Carlo Ghezzi and Giovanni Vigna. Mobile code paradigms and technologies: A case study. In *Proceedings of the First International Workshop on Mobile Agents*, Berlin, Germany, April 1997.

[40] Robert S. Gray, David Kotz, George Cybenko, and Daniela Rus. Mobile agents: Motivations and state-of-the-art systems. Technical Report TR2000-365, Dartmouth College, Hanover, NH, 2000.

[41] Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, and Henrik Frystyk Nielsen. Soap version 1.2 specification, June 2003.

[42] L. Hagen, M. Breugst, and T. Magedanz. Impacts of Mobile Agent Technology on Mobile Communications System Evolution. *IEEE Personal Communications*, 5(4), August 1998.

[43] Steve Hanks, Martha E. Pollack, and Paul R. Cohen. Benchmarks, testbeds, controlled experimentation, and the design of agent architectures. *AI Magazine*, 14(4):17–42, 1993.

[44] Patrick Hayes. Rdf semantics, February 2004.

[45] Marc-Philippe Huget. Agent uml notation for multiagent system design. *IEEE Internet Computing*, 8(4):63–71, 2004.

[46] Michael N. Huhns and Munindar P. Singh, editors. *Readings in Agents*. Morgan Kauffman Publishers, San Francisco, CA, 1998.

[47] IEEE. Standard upper ontology working group, November 2006. http://suo.ieee.org/.

[48] Nicholas R. Jennings, Katia Sycara, and Michael Woolridge. A roadmap of agent research and development. *Journal of Autonomous Agents and Multi-Agent Systems*, 1(1):7–38, 1998.

[49] David Kotz, Robert Gray, and Daniela Rus. Future directions for mobile-agent research. Technical Report TR-2002-415, Dartmouth, Hanover, NH, Januray 2002.

[50] Philippe Kruchten. Architectural blueprints—The "4+1" view model of software architecture. *IEEE Software*, 12(6):42–50, November 1995.

[51] Yannis Labrou and Tim Finin. A Proposal for a new KQML Specification. Technical Report TR CS-97-03, Computer Science and Electrical Engineering Department, University of Maryland Baltimore County, Baltimore, MD, USA, February 1997.

[52] Alberto Leon-Garcia and Indra Widjaja. *Communication Networks*, chapter 5, page 267. McGraw Hill, 2000.

[53] Victor R. Lesser. Evolution of the GPGP/TæMS domain-independent coordination framework. In Maria Gini, Toru Ishida, Cristiano Castelfranchi, and W. Lewis Johnson, editors, *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'02)*, pages 1–2. ACM Press, July 2002.

[54] Raphael Malveau and Thomas J. Mowbray. *Software Architect Bootcamp*. Prentice Hall, 2001.

[55] Dr. Israel Mayk and Bernard Goren. C2 product-centric approach to transforming current c4isr information architectures.

[56] Deborah L. McGuinness and Frank van Harmelen. Owl web ontology language, February 2004.

[57] Dejan Milojicic, Frederick Douglis, and Richard Wheelter. *Mobility: Processes, Computers, and Agents*. Addison-Wesley, 1999.

[58] Hyacinth S. Nwana. Software agents: An overview. *Knowledge Engineering Review*, 11(3):1–40, September 1996.

[59] J. Postel. User Datagram Protocol. RFC 768 (Standard), August 1980.

[60] J. Postel. Transmission Control Protocol. RFC 793 (Standard), September 1981. Updated by RFC 3168.

[61] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Upper Saddle River, NJ, 2 edition, 2003.

[62] Alberto Rodrigues Silva, Arthur Romao, Dwight Deugo, and Miguel Mira Da Silva. Towards a reference model for surveying mobile agent systems. In *Autonomous Agents and Multi-Agent Systems*, pages 187 – 231, 2001.

[63] Niranjan Suri. Nomads and agile computing. Presentation at IPT Meeting.

[64] K. Sycara, J. Lu, M. Klusch, and S. Widoff. Dynamic service matchmaking among agents in open information environments. In *Journal ACM SIGMOD Record, Special Issue on Semantic Interoperability in Global Information Systems*, 1999.

[65] Katia P. Sycara. Multiagent systems. *AI Magazine*, 19(2):79–92, Summer 1998.

[66] Mary Ann C. Tuckman, Bruce W.; Jensen. Stages of small group development revisited. *Group and Organization Studies*, 2:419–426, December, 1977.

[67] International Telecomm Union and the International Organization for Standardization. Draft: Common logic (cl) – a framework for a family of logic-based languages, September 2005.

[68] Giovanni Vigna, editor. *Mobile Agents and Security*. Lecture Notes in Computer Science. Springer-Verlag, New York, NY, 1999.

[69] Gerhard Weiss. *Multiagent Systems: A Modern Approach to Distsributed Artificial Intelligence*. MIT Press, Boston, MA, 2001.

[70] Michael Wooldridge. *Reasoning about Rational Agents*. MIT Press, Boston, MA, 2000.

[71] Michael Wooldridge. *Introduction to Multi-Agent Systems*. Wiley & Sons, New York, NY, 2002.

[72] Todd Wright. Naming services in multi-agent systems: A design for agent-based white pages. In *Proceedings of Third International Joint Conference on Autonomous Agents and Multiagent Systems*, 2004.

[73] Dianxiang Xu, Jianwen Yin, Yi Deng, and Junhua Ding. A formal architectural model for logical agent mobility. *IEEE Trans. Softw. Eng.*, 29(1):31–45, 2003.

[74] Herbert Zimmerman. OSI reference model—the ISO model of architecture for open system interconnection. *IEEE Transactions on Communications*, 28(4):425–432, April 1980.

Index