A REFERENCE MODEL FOR AGENT-BASED COMMAND AND CONTROL SYSTEMS

*Christopher J. Dugan, Pragnesh Jay Modi, Joseph B. Kopena, William M. Mongan, and William C. Regli
Drexel University
Philadelphia, PA, 19104

Israel Mayk
C2 Directorate (C2D)
US Army Research, Development and Engineering Command (RDECOM)
Communications-Electronics Research, Development and Engineering Center (CERDEC)
Fort Monmouth, NJ 07703

ABSTRACT

Standardization and the ability to integrate similar agent-based systems will be a key factor in the deployment of future military agent-based systems: multiple developers can coordinate in the development phase, integration with existing components will be streamlined in the deployment phase, and collaboration with similar systems (such as coalition forces) will be trivial. Unfortunately, there exists no taxonomy of terms that can describe concepts, definitions, and functional elements within agent-based systems. This makes accomplishing the above difficult—if not impossible. In this paper, we describe a reference model for agent-based systems and show the methodology behind its development. Such a comprehensive reference model will facilitate adoption, adaptation, and integration of agent technologies into systems for use by government and private industry, with a particular focus on applications in military command and control.

1. Introduction

A reference model describes the abstract functional elements of a system. It does not impose specific design decisions on a system designer. APIs, protocols, *etc.* are standards that can be used concurrently with a reference model. A reference model does not define an architecture. Multiple architectures can be derived from a reference model in the same way that reference architectures could drive multiple designs or a design could drive multiple implementations (see Figure 1). As such, a reference model

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

Currently, no reference model exists that describes agent systems—making it impossible to uniformly describe a system in the aforementioned terms. Thus, we developed the Agent Systems Reference Model (ASRM) that aims to allow for existing and future agent frameworks to be compared and contrasted, as well as to provide a basis for identifying areas that require standardization within the agents
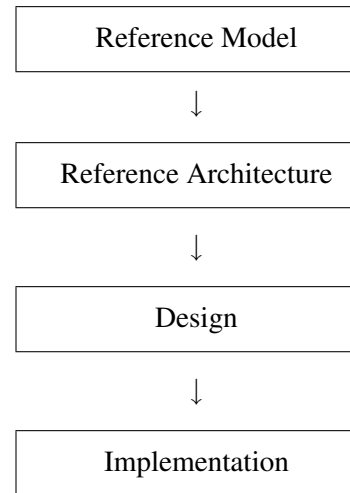

Figure 1: A reference model drives creation of one or more reference architectures, which drive the creation of one or more designs, which drives the development of one or more implementations.

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.

1.1 Motivation

The motivation for this project is in part analogous to the previous need for a communications standard, 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 (Zimmerman, 1980). A 7-layer model, the Open Systems Interconnection (OSI) reference 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 frameworks exist that can lay the groundwork for such an agent systems reference model.

1. REPORT DATE 01 NOV 2006	2. REPORT TYPE N/A	3. DATES COVERED -

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
A Reference Model For Agent-Based Command And Control Systems	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Drexel University Philadelphia, PA, 19104**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT **Approved for public release, distribution unlimited**

13. SUPPLEMENTARY NOTES **See also ADM002075., The original document contains color images.**

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**UU**	**8**	

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std Z39-18

1.2 Approach

The ASRM needs to describe several elements of an agent system, including agents and frameworks. An agent is obviously a building block of any agent system; however, the ASRM does not develop a consensus about the definition of an agent due to the largely inconclusive debates of the past. There are many products in the marketplace today that are marketed as agent frameworks from various sources: companies, academia and the open source community. These agent frameworks have emerged from several large governmental and private research and development programs and were used in the creation of many successful military and commercial systems. The ASRM takes a quantitative and evidentiary approach; if it can be built with one of these systems, an artifact might be called an "agent." Likewise, static and dynamic analysis of these agents, their frameworks, and other related entities is used to create the ASRM. For a more detailed discussion of these entities and their purpose, see Section 2. See Section 4 for more details about how the reference model was created.

1.3 Related Efforts

In the area of agent systems, a reference model for mobile agent systems was constructed in (Silva et al., 2001). 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, it presents a set of minimum feature requirements. That is, the *Agent Execution System* is a required component that supports mobility, communication, agent serialization and security.

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

2. Agent System Concepts and Layers

The core of the ASRM is the layered diagram portrayed in Figure 2. All agent systems can be mapped to this diagram—an important characteristic of any reference model. This figure certainly introduces some ambiguous terms which will be defined in the following sections.

2.1 What is an Agent?

Software agents, sometimes called intelligent agents or simply "agents," are situated computational processes—instantiated programs that exist within an environment which they sense and effect. An agent actively receives percepts, signals from the environment, through a sensor interface. Though its response need not be externally ob-
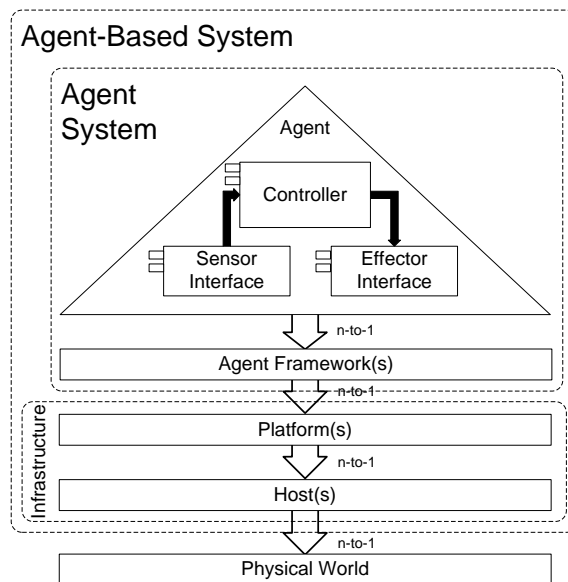

Figure 2: Abstract model of an agent system. Such systems decompose into several layers of hardware and software that provide an operating context for agents.

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

Note the distinction between agents and services. Services are computational processes that exist to provide functionality for use by other processes. Services are infrequently associated with autonomy and proactivity, but seem to be interactive. We claim that services are not interactive in the sense that they are not bound to their external environment; whereas, agents are more dependent on their environment.

2.2 Agent-Based Systems

As computational processes, agents do not exist on their own but rather within computing software and hardware that provides them mechanisms to execute. Many agent implementations also require substantial libraries and code modules. Further, agents frequently possess prop-

erties 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 that supports them. It is comprised of several layers as shown in Figure 2.

Agent Layer. Agents implement the application and they achieve the intended functionality of the system. A more in depth description can be found above.

Framework Layer. 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 a collection of system calls or is compiled into the agents themselves.

Platform Layer. The platform provides 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.

Host Layer. 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.

Physical World Layer. This 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-based system is simply a set of frameworks and agents that execute in them. A multi-agent system is an agent-based system which 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.* Figure 3 shows an 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. Note the distinction between instantiation and type.

2.3 MAS Structure

A set of more than one agent will be collectively referred to as a group. In addition, a set of groups may also be referred to as a group. Then, the term 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
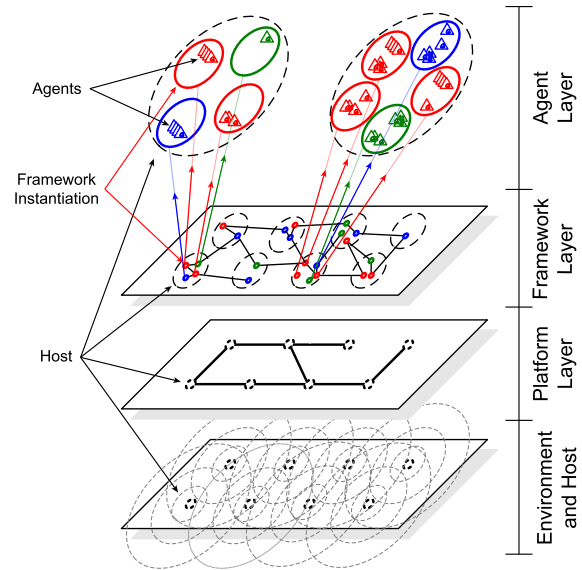

Figure 3: Agents are depicted as computational processes running within frameworks supported by platforms and executing on hosts operating together on a wireless network.

complexity.

There exist three basic types of MAS. Monolithic systems involve a single agent of high internal complexity. An example would be a proxy agent which conducts tasks for a user such as scanning the World Wide Web for prices and making purchases to fill given specifications. Median systems contain moderately complex and heterogensous agents. Often, these types of systems involve coordination, cooperation, and resource sharing in order to achieve a common goal. Robot soccer is a good example of a median system. Finally, swarm systems involve many similar agents of low internal complexity. Swarms are studied because interesting behaviors result from the large number of agents interacting with each other. They provide a large degree of redundancy and the ability to introduce even more agents with relative ease. An example of a swarm is the modeling of a traveling salesman problem using ant agents who leave pheromone to make statistical decisions.

2.3.1 Structured Groups of Agents

Groups have the property that they can be specialized. In such a case, all of the agents in a group have a common goal or perform a common task. 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.

- An *organization* is a group that interacts according to some structure, such as a hierarchy. Each agent or group has a goal which 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 which 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. Functional Concepts

Agent systems can be viewed as a set of abstract concepts that support overall system execution. For example, security and mobility are two abstract functional concepts (among others) described. It is important to note that these descriptions make 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.

While a brief overview of functional concepts is given below, a more detailed description can be found at (Modi et al., 2006).

3.1   Agent Administration

Agent administration functionality facilitates and enables supervisory command and control of agents and/or agent populations and 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*.

The functional concepts that fall under the heading of agent administration include:

- *Agent Creation.* The act of instantiating or causing the creation of agents.

- *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.

- *Resource Control.* The process by which an agent's access to system resources is controlled.

- *Agent Termination.* The process by which agents are terminated (*i.e.*, their execution is permanently halted).

3.2   Security and Survivability

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 purpose of survivability is for the system to be useful while remaining dependable in the face of malice, error or accident.

The functional concepts that fall under the heading of agent administration include:

- *Authentication.* A process for identifying the entity requesting an action.

- *Authorization.* A process for deciding whether the entity should be granted permission to perform the requested action.

- *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.

3.3   Mobility

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

The functional concepts that fall under the heading of agent administration include:

- *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. By contrast, active mobility occurs when the agent is in control of its own mobility, and decides when and where it shall migrate on its own.

- *De-register, Halt, Serialize.* Once an agent has decided (or been notified) that it is migrating, it must de-register from all of the directory services on the framework instantaition with which it has registered. Then, it halts execution, and is serialized. The serialization process involves persisting the agent's data and/or state into a data structure. This data structure is converted to packets or written to a buffer to prepare the agent for migration.

- *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 has left the host; instead, the agent is changing the framework instance on which it is executing. Recall that a host and platform may be housing multiple framework in-

4

stances, allowing for migration within a particular host. According to (Xu et al., 2003), 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.

- *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. Then, 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.

3.4 Conflict Management

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. 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.

The functional concepts that fall under the heading of agent administration include:

- *Conflict avoidance.* A process or mechanism for preventing conflicts.
- *Conflict detection.* The process of determining when a conflict is occurring or has occurred.
- *Conflict resolution.* The process through which conflicts between agent activities are resolved. Negotiation, mediation and arbitration are common mechanisms for facilitating conflict resolution.

3.5 Messaging

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. 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 con-

duct messaging. However, messaging is defined here as a stand-alone concept of its own right. Some areas of interest in messaging functionality include notions of best effort delivery, QoS and guaranteed delivery/timeliness.

The functional concepts that fall under the heading of agent administration include:

- *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.
- *Naming and Addressing.* A mechanism for labeling the message with its intended destination or route.
- *Transmission.* The actual transport of the message over the channel. This may be a one-shot transmission or a continuous stream.
- *Receiving.* The process for acquiring the transmitted information so that is usable by the receiver.

3.6 Logging

Logging functionality facilitates and enables information about events that occur during agent system execution to be retained for subsequent inspection. This includes but does not imply persistent long-term storage. Logging is a supporting service that provides informational, debugging or management information about the agent system as it executes. It can be a centralized service or distributed amongst the agents (wherein each agent performs its own logging).

The functional concepts that fall under the heading of agent administration include:

- *Log Entry Generation.* The process by which information to be logged is created. Log entries often include type (informational, warning, critical, among others) or priority (for instance, priority 1 through 5).
- *Storing Log Entry* Log entries are stored in a variety of ways at the choosing of the implementation of the agent system.
- *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.

3.7 Directory Services

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.

The functional concepts that fall under the heading of agent administration include:

- *Naming* The process by which resources are assigned identifiers so that they may be indexed and located.

5

This process can be fairly complex by supporting group names, transport addresses, dynamic name resolution, and other complex features (Wright, 2004).

- *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 an access method.

- *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) (Sycara et al., 1999).

4. 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. In creating the ASRM, reverse engineering and software analysis methods are employed. It is necessary 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.

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 that exist 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.

4.1 Documenting the Reference Model: The 4+1 Model

According to (Kruchten, 1995), 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 compo-

nent 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

- *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.

4.2 Reverse Engineering Techniques for Informing a Reference Model

Reverse Engineering 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 that exist between components. It is often the case in software systems that components are relatively loosely coupled, but are locally tightly coupled. In other words, most components do not depend directly on one another on the whole, while related components interact to achieve their common functionality.

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 *in-*

ternal 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 components with only a high degree of coupling. This process may be repeated any number of times by further grouping entire clusters based on their coupling. Software analysis tools exist to extract and 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 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.

4.3 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, 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.

4.4 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.

4.5 Command and Control (C2) and the ASRM

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 (Tuckman and Jensen, 1977). Each process performs this cycle asynchronously, applying and sharing aggregated information appropriately and 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.

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. 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.

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 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. 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.

The following describes 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 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. The processes, subprocesses and phases of Command and Control interoperate. 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 Federate phase corresponds to the Tuckman form stage.

- *Federate (Initialize).* This represents the higher level ISO communication layers in which a protocol is cho-

sen 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, so it is used by all the other processes within C2.

5. Conclusion

A reference model for agent-based systems produces a taxonomy of terms that can describe concepts, definitions, and functional elements within and between individual systems. We believe the ASRM accomplishes such a task. We have shown that agent systems are relevant to the Command and Control domain. Once a reference model is agreed upon, C2 military applications can be built and integrated with relative ease. Further, these applications can interoperate with existing applications allowing the military or a coalition force to perform cooperatively as was never previously possible.

References

Kruchten, P., 1995: Architectural blueprints—The "4+1" view model of software architecture. *IEEE Software*, **12**, 42–50.

Modi, P., S. Mancoridis, W. Mongan, W. Regli, and I. Mayk, 2006: Towards a reference model for agent-based systems. *Proceedings of Autonomous Agents and Multiagent Systems, Industry Track*.

Silva, A. R., A. Romao, D. Deugo, and M. M. D. Silva, 2001: Towards a reference model for surveying mobile agent systems. *Autonomous Agents and Multi-Agent Systems*, 187 – 231.

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

Tuckman, B. and M. Jensen, 1977: Stages of small group development revisited. *Group and Organization Studies*, **2**, 419–426.

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

Xu, D., J. Yin, Y. Deng, and J. Ding, 2003: A formal architectural model for logical agent mobility. *IEEE Trans. Softw. Eng.*, **29**, 31–45, doi:http://dx.doi.org/10.1109/TSE.2003.1166587.

Zimmerman, H., 1980: OSI reference model—the ISO model of architecture for open system interconnection. *IEEE Transactions on Communications*, **28**, 425–432.