2011

Coordination Of Hierarchical Command And Control Services

Srinivas Reddy Banda
North Carolina Agricultural and Technical State University

Coordination of Hierarchical Command and Control Services

Srinivas Reddy Banda

North Carolina A&T State University

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Department: Computer Science and Engineering

Major: Computer Science

Major Professor: Dr. Albert C Esterline

Greensboro, North Carolina

2011

School of Graduate Studies
North Carolina Agricultural and Technical State University

This is to certify that the Master's Thesis of

Srinivas Reddy Banda

has met the thesis requirements of
North Carolina Agricultural and Technical State University

Greensboro, North Carolina
2011

Approved by:

Dr. Albert C Esterline Major Professor	Dr. Gerry V Dozier Committee Member
Dr.Jinsheng Xu Committee Member	Dr. Gerry V Dozier Department Chairperson

Dr. Sanjiv Sarin
Associate Vice Chancellor for Research and Graduate Dean

Biographical sketch

Srinivas Reddy Banda was born in Hyderabad, India on April 30, 1988. He received his Bachelor of Engineering degree in 2009 in the Department of Computer Science and Engineering at the Jawaharlal Nehru Technological University, Hyderabad, India. Then, he joined North Carolina Agricultural and Technical State University in January 2010, where he is currently a candidate for Master of Science and a research assistant in the department of Computer Science.

Dedication

This thesis is dedicated to my parents, Chandra Shekar Reddy Banda and Manjula Banda for their love, support and encouragement.

Acknowledgments

I would like to express the deepest appreciation to my advisor, Dr. Albert C Esterline, who convincingly conveyed a spirit of adventure in regard to the entire idea of learning and research. His enthusiasm and guidance were the most important factors that have made this work possible today.

I am also thankful to Dr. Dozier and Dr. Xu for accepting to be my committee members and providing me with valuable advice and guidance to make this dissertation possible. I would also like to thank William Wright for providing an insight in all the discussions we undertook during this research. I am also particularly indebted to the direction and assistance generously provided by Dr. Israel Mayk of the US Army CERDEC C2D.

Last but not the least; I am very thankful to my family and friends who have always been a constant source of encouragement and support which enabled me to maintain the energy levels required to keep working on this research.

Table of Contents

List of Figures

Abstract

An Advanced Technology Objective (ATO) funded by the US Army is the Tactical Information Technologies for Assured net Operations program. The purpose of this program is to show emerging information technologies can significantly improve key areas of tactical operations, resulting in the conversion of software developed under the ATO to existing battlefield systems. One such key area is Information Dissemination and Management (ID&M). The key software that will be developed under the ID&M portion requires a collection of agent-based software services that will collaborate during tactical mission planning and execution.

The agent framework used here is JADE. This work implements prototypes for three TITAN services, the Alert and Warning Service (AWS), the OPORD Support Service (OPS), and the Workflow Orchestration Support (WOS). The work reported here integrates multiagent systems, Web services, distributed event-based systems (in the form of JMS) and Workflows (WADE) for collaboration and dissemination of information. The information used refers to military command and control information represented in XML documents. The integration uses gateways provided as add-ons to the agent framework that is used.

Common Knowledge is a prerequisite for coordination among human or artificial agents. Since most of the environments are dynamic, the common knowledge agents what initially have generally is not sufficient for agents to coordinate their actions in each and every situation. Theory of mind is the ability to attribute mental states (beliefs, desires, intentions, assumed roles, etc.) to one and others and to understand that others have mental states different from one's own. Common knowledge and Theory of mind are considered for the coordination of agents in TITAN suites.

CHAPTER 1

Introduction

An Advanced Technology Objective (ATO) funded by the US Army is the Tactical Information Technologies for Assured net Operations (TITAN) (US Army, 2008) program. The purpose of this program is to exhibit how emerging information technologies can significantly improve key areas of tactical operations, finally resulting in the conversion of software developed under the ATO to existing battlefield systems. One such key area is Information Dissemination and Management (ID&M). The key software that will be developed under the ID&M portion of the TITAN Program requires a collection of agent-based software services that will collaborate during tactical mission planning and execution. The agent framework that will be used is JADE(Bellifemine, Caire, & Greenwood, 2007). The other technologies that will be used, besides multiagent systems, are rule-based systems, distributed event based systems, web services, and workflows.

The work done is part of a small effort being pursued at North Carolina A&T State University that parallels a larger effort supported by the Command and Control Directorate (C2D) of the Communications-Electronics Research, Development, and Engineering Center (CERDEC) of the US Army. The small effort has been supported since the spring of 2007 by the C2D and so far resulted in four Computer Science theses by Barnette (2008); Krishnamurthy (2010); Mason (2008); Reid (2007). The larger effort aims at implementation of the TITAN services using Cougaar(Kappler, 2009) as agent framework, and the work here is the smaller effort that implements part of TITAN using JADE which is an alternative to Cougaar. The three TITAN services implemented are Alert Warning System (AWS), OPORD Support Service (OPS) and Workflow Orchestration Support Service (WOS). Figure 1 shows the three paradigms

used for collaboration and information dissemination. These paradigms are considered here as the sides of a triangle. Technologies to integrate these paradigms in pairs are shown on the vertices.

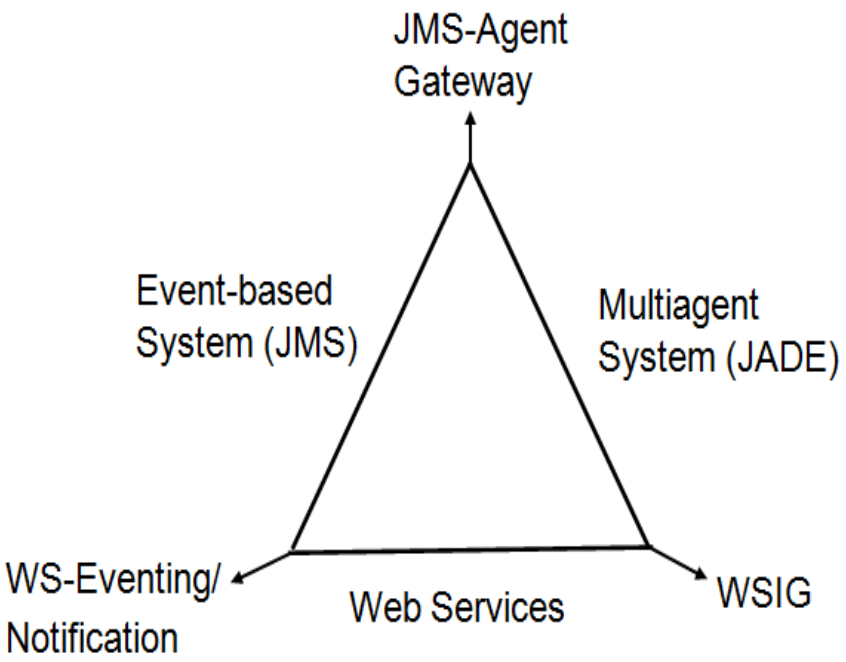

Figure 1. Three integrating technologies for information dissemination (Krishnamurthy, 2010).

Common Knowledge is a prerequisite for coordination among human or artificial agents. Since most of the environments are dynamic, the common knowledge agents what initially have generally is not sufficient for agents to coordinate their actions in each and every situation. Theory of mind is the ability to attribute mental states (beliefs, desires, intentions, assumed roles, etc.) to one and others and to understand that others have mental states different from one's own. Common knowledge and Theory of mind are considered for the coordination of agents in TITAN suites.

The remainder of this thesis is ordered in as follows. The next chapter explores different technologies used for the implementation of this project. The major technologies incorporated in this project are distributed event-based systems, Web services, multi-agent systems (along with

rule-based systems) and workflows. The next chapter also introduces the principle analytic concept addressed in the discussion, namely, common knowledge. The third chapter discusses the TITAN program. The fourth chapter discusses the implementation of the TITAN prototype. The sixth chapter is discusses the conditions for coordination in the model of TITAN in terms of common knowledge; it also summarizes the advantages of the technologies used and their integration. The final chapter concludes and suggests future work.

CHAPTER 2

Background

This chapter introduces the required information and technologies for understanding the work proposed. It has several sections, which describe much of the background in general, including multiagent systems, Web services, distributed event based systems, and workflows. There is also a JADE WSIG add-on that allows interoperability among multiagent systems and Web services. The discussion of multiagent systems also addresses the Foundation for Intelligent Physical Agents (FIPA), an IEEE standards organization for agents, the Java Agent Development Framework (JADE), which conforms to FIPA standards, and the Java Expert System Shell (JESS), used in conjunction with JADE. The discussion of distributed event based systems also covers Java Messaging Service (JMS), which is an API for messaging, and the JMS Agent Gateway, which makes JMS functionality available to JADE agents. XML is the standard language for communicating data on the Web and is the language through which data is communicated among TITAN services. Java Architecture for XML Binding (JAXB) is a Java API that allows marshaling of Java objects into XML documents and unmarshalling XML documents into Java objects. Workflow and Agents Development Environment (WADE) adds to JADE functionality for execution of tasks that are defined as workflows.

2.1 Multiagent System

Multiagent systems (MASs) are systems composed of multiple interacting computing elements known as *agents.* In general, agents are computer systems that are capable of autonomous action and interact with other agents.

According to Wooldridge (2009) which is the major source of material for this section, an intelligent agent should possess three characteristics. It is reactive, is proactive, and has social

ability. An agent is reactive if it is able to perceive its environment and respond in a timely fashion to changes that occur in it. An agent is proactive if it exhibits goal-directed behavior by taking the initiative. An agent has social ability if it is capable of interacting with other agents (and possibly humans). Agents mutually interact with each other and form a multiagent system (MAS). In MAS, each agent has different goals but they all collaborate to achieve the goals of the system.

To allow agents to communicate with each other, several agent communication languages (ACLs) have been defined. An ACL is typically grounded in speech-act theory, which` analyzes language use as consisting of acts, such as asserting, requesting, or commanding (Bellifemine et al., 2007). Some verbs, such as “request”, “assert” and “command,” represent actions done with words and are called performatives. An older ACL for MASs is KQML (Knowledge Query and Manipulation Language), which was developed as part of the ARPA Knowledge Sharing Initiative (Milner, 1989). KQML is intended for large-scale knowledge bases that are shareable and reusable (JADE-Board). A more recent (and, from a multi agent point of view, improved) ACL is FIPA-ACL, which was developed by FIPA.

2.1.1. Agent development frameworks. Agent development frameworks provide middleware that helps in developing multiagent systems. Many frameworks have been developed in recent years. Two categories of such frameworks are those for developing agents based on the Beliefs-Desires-and-Intentions (BDI) agent model and those for developing agents viewed as holons.

This section reviews these two categories of agent models and frameworks, and then it reviews an important framework not in either category. The following sections introduce the FIPA standards body for agent technology and outline JADE, the agent framework to be used in

the proposed research.

“Holon”(Koestler, 1968) combination of the Greek words “holos,” which means whole, and the suffix “on,” which means part. A holarchy is defined as a hierarchy of holons. Koestler viewed holons as filling roles and as defined by fixed rules and flexible strategies, as best illustrated perhaps by language, which has fixed grammatical rules yet supports endless forms of conversation (Esterline, BouSaba, Pioro, & Homaifar, 2006). Holonic Manufacturing is based on the concept of “holonic systems”, developed by Koestler. In engineering domains, holarchies are used mostly in holonic manufacturing systems (HMSs), providing ways to organize manufacturing systems so that the key elements (i.e., holons), such as machines, plants, products, personnel, and departments, have autonomous and cooperative properties (Babiceanu & Chen, 2006).

Generalizing beyond engineering domains, holons are used in holonic multiagent systems (HMASs). An HMAS considers agents as holons that are grouped according to holarchies (Hilaire, Koukam, & Rodriguez, 2008). The HMAS perspective provides terminology and theory for the realization of dynamically organizing agents. This perspective attributes modularity and recursion to the agent paradigm. In an HMAS, an agent that appears as a single entity to the outside world may in fact be composed of many sub-agents, and, conversely, many subagents may decide that it is advantageous to join into the coherent structure of a super-agent and thus act as single entity (Schillo & Fischer, 2003).

One of the holonic agent development frameworks is Janus (Multi agent Group), an open-source multi-agent platform fully implemented in Java. Janus is built upon the Capacity, Role, Interaction and Organization (CRIO) organizational meta-model and supports the implementation of the concepts of role and organization as first-class entities. It also natively

manages the concept of recursive agents or holons.

The Belief-Desire-Intention (BDI) software model (Bratman, Israel, & Pollack, 1988) is a model for developing intelligent agents. The roles played by attitudes such as beliefs (B), desires (D), and intentions (I) in the behavior of rational agents has been well recognized in the philosophical and AI literature (Bratman, 1990; Bratman, Israel, & Pollack, 1987; Bratman et al., 1988).

Figure 2 represents architecture for practical reasoning in resource bounded agents. It can be classified as BDI architecture. It includes direct representation of an agent's beliefs, desires and intentions. The agent's intentions are structured into large plans. The plan that the agent has actually adopted, which is represented by the oval labeled "Intentions structured into Plans" in figure 2, and plans-as-recipes that agent already knows, represented by the oval labeled "Plan Library," are distinguished. The Plan Library can be viewed as a subset of the agent's beliefs. Besides information stores (represented by ovals), there are processes (denoted by rectangles). These include the "Means-End Reasoner", the "Opportunity Analyzer", the "Filtering Process" and the "Deliberation Process". Together these constitute a practical-reasoning system by which an agent forms, fills in, revises and executes plan. Means-end Reasoner and Deliberation are discussed in the later part of this section. The architecture depicted in the Figure 2 is an account of functional roles of an agent's plan not just in producing action, but also practical reasoning– an account so far developed by Bratman. In this, an agent's existing plans make practical reasoning in two ways: as input to the means-end reasoner and as input to the filtering process.

It is possible to distinguish several types of intentions. In ordinary speech, "intention" is used to characterize actions and states of mind, but, whenever "intention" is used in this proposal, it refers to a state of mind. Particularly important are future-directed intentions –

intentions that an agent has towards some future state of affairs. The most apparent role of intentions is as *pro-attitudes* (Bratman, 1990), that is, as they tend to lead to action.

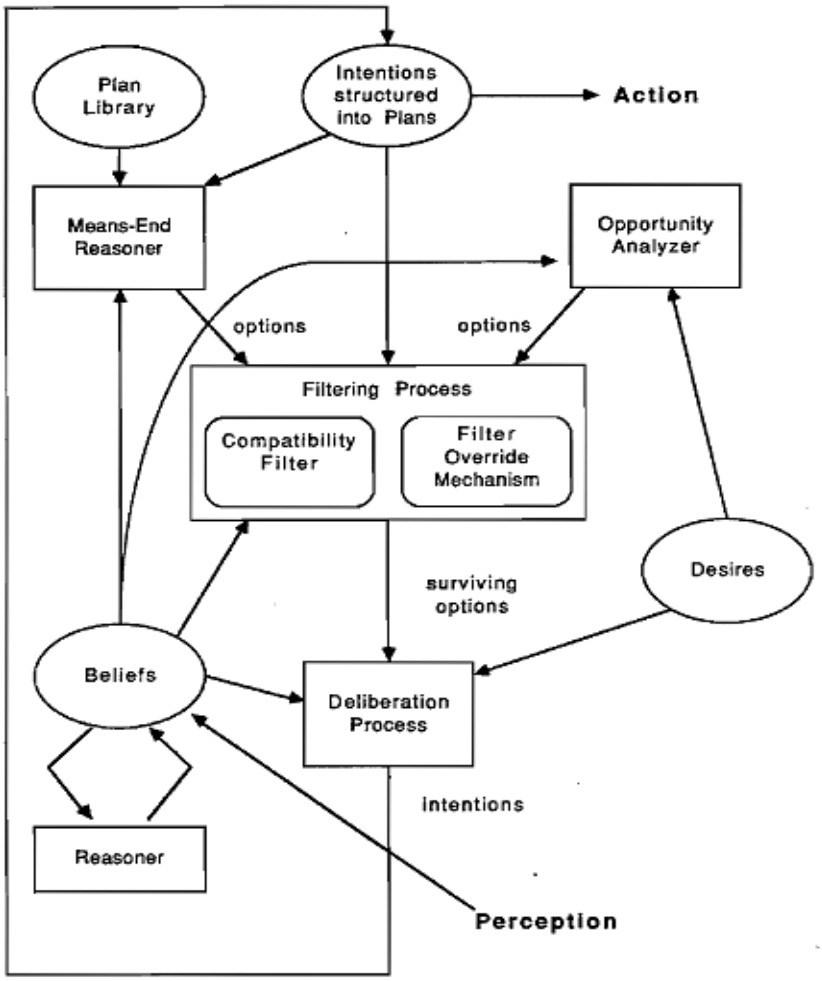

Figure 2. Belief-Desire-Intention Architecture (Bratman et al., 1988).

The second main property of intentions is that they *persist.* If someone adopts an intention to become an academic, for example, then they should persist with this intention and attempt to achieve it. For, if they immediately drop their intentions without devoting any resources to achieving them, then they will not be acting rationally. Indeed, we might be inclined to say that they never really had intentions in the first place. The third main property of

intentions is that, once one has adopted an intention, the very fact of having this intention will constrain one's future practical reasoning. The final property is that intentions are closely related to beliefs about the future. Intentions and desires are both pro-attitudes, but intentions are conduct-controlling while desires are not (Wooldridge, 2009).

Human practical reasoning seems to have at least two distinct activities. The first involves deciding what state of affairs to be achieved; the second involves deciding how to achieve these states of affairs. The former process—deciding what states of affairs to achieve—is known as deliberation. The latter process—deciding how to achieve these states of affairs—is called means-ends reasoning (Wooldridge, 2009).

JACK (AOS Group) is an agent framework based on the BDI software model. A JACK application consists of a collection of autonomous agents that take input from the environment and communicate with other agents. JACK provides system builders with a very powerful form of encapsulation. Each agent is defined in terms of its goals, knowledge and social capability and is then left to perform its function autonomously within the environment it is embedded in. JACK is a mature implementation of the BDI paradigm and has a sophisticated graphical development environment that supports the design, implementation and monitoring of BDI agents.

There are many agent frameworks based on the BDI model. The Procedural Reasoning System (PRS) (Myers) was the first agent architecture based on the BDI paradigm. Jason (Hübner & Bordini) is an interpreter for an extended version of AgentSpeak. It implements the operational semantics of that language and provides a platform for the development of multiagent systems, with many user-customizable features. The SRI Procedural Agent Realization Kit (SPARK) (SPARK Team) is a new agent framework, under development at the

Artificial Intelligence Center of SRI International. SPARK builds on the success of its predecessor, PRS, and shares the same BDI model of rationality, 3APL (An Abstract Agent Programming Language) (Dastani), developed by the Intelligent Systems Group at the University of Utrecht, is a programming language for implementing cognitive agents. It provides programming constructs for implementing agents' beliefs, goals (or intentions), and basic capabilities. 2APL (A Practical Agent Programming Language) (Dastani) is the successor to 3APL and is an agent-oriented programming language that facilitates the implementation of multi-agent systems. At the multiagent level, it provides programming constructs to specify a multiagent system in terms of a set of individual agents and a set of environments in which they can perform actions. The Cognitive Agent Architecture (Cougaar) (Cougaar Team, 2004) project focuses on developing and maintaining open source agent architecture for the construction of large-scale distributed agent-based applications. It is based on neither the holonic nor the BDI paradigm. Cougaar is sponsored by the Defense Advanced Research Projects Agency (DARPA) (US Army).

2.1.2. Foundation of intelligent physical agents. It is an IEEE Computer Society standards body for agent technology. FIPA specifications represent a collection of standards that help in interaction and interoperation between heterogeneous agents (FIPA Board).

The FIPA standards committee (SC) has two kinds of groups, called working groups (WGs) and study groups (SGs) (FIPA Board). It has six working groups: Agents and Web Services Interoperability Working Group (AWSI WG), Design Process Documentation and Fragmentation Working Group (DPDF WG), Application Specifications Working Group (AS WG), Human-Agent Communications Working Group (HAC WG), P2P Nomadic Agents Working Group (P2PNA WG) and Mobile Agents Working Group (MA WG). Specifically, the

main objective of AWSI WG is to fill the interaction gap between Web services and agents: agents should be able to locate and interact with web services.

2.1.3. Java agent development framework. Java Agent Development Framework (JADE) is a software framework that complies with FIPA specifications (Bellifemine et al., 2007). It simplifies the implementation of multiagent systems. The basic middleware functionalities of the platform are independent of the specific application and simplify the realization of distributed applications exploiting the software agent abstraction. A considerable pro of JADE is that it implements this abstraction in Java with a simple and friendly API.

Each and every agent needs its own thread to control its life cycle and to decide which action to perform and when since agents are autonomous and proactive. To prevent an agent from co-opting control of its service to another agent, it should not provide call backs or references to the other agent.

A JADE platform consists of several agent containers that are located over the network. Agents live in containers that provide JADE runtime support and all the services needed for hosting and executing agents. The main container, called “Main Container,” is the first container launched. All other containers join the main container by registering themselves with the main container.

Figure 3 shows the relationship between the main elements of a JADE platform: the Agent Management System (AMS), the Directory Facilitator (DF), the Message Transport Protocol (MTP), Main Container, two additional containers (called Container-1 and Container-2), the Local Agent Descriptor Tables (LADTs), the Global Agent Descriptor Tables (GADTs), and the Container Table (CT). The Directory Facilitator (DF) by default is in Main Container, and is the agent providing yellow-pages service. The AMS is always in Main Container and it

supervises the entire platform, manages the life-cycles of all agents, and provides white-pages service. JADE has every agent register with the AMS to obtain an AID (Agent Identifier).

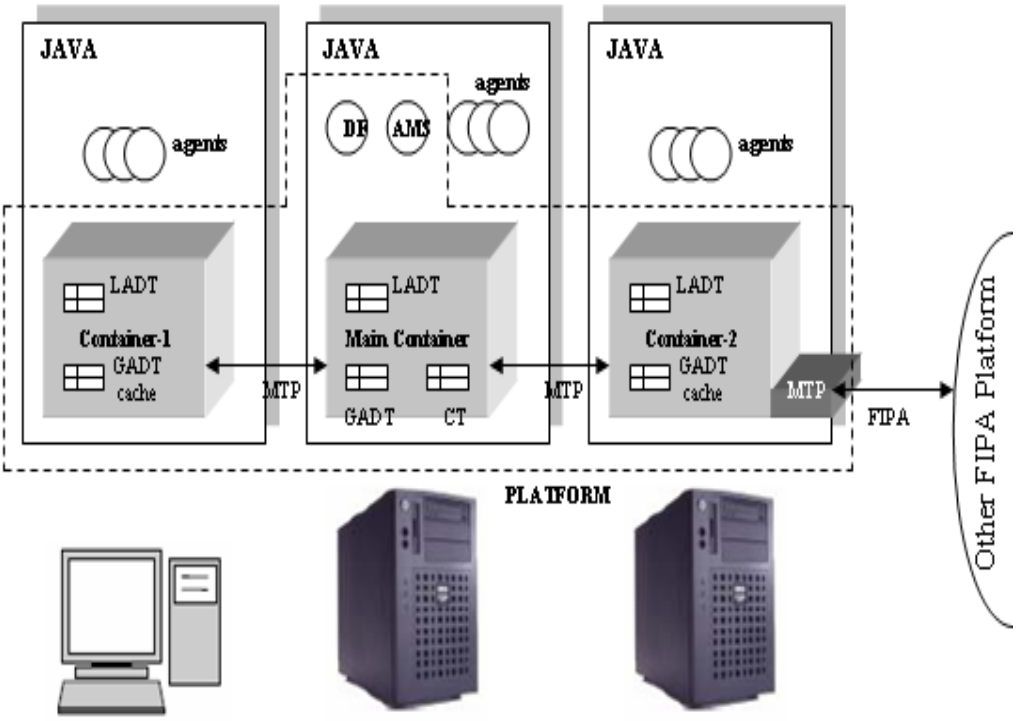

Figure 3. JADE agent platform architecture (Bellifemine et al., 2007).

According to the FIPA standard specifications, the MTS (Message Transport Service) is one of the most important services out of the three (including the AMS and DF) an agent platform is required to provide. All message exchanges are performed by it within and between platforms. Numerous MTPs can be activated on a given JADE container. The LADT (Local Agent Descriptor Table) for a given container is the registry of all agents in that container whereas the GADT (Global Agent Descriptor Table) in Main Container is the registry of all agents in the platform; to prevent the main container from becoming a bottleneck, JADE provides a cache of the GADT in each container. The CT (Container Table) is the registry of the object references and transport addresses of all container nodes in the agent platform.

2.1.4. Java expert system shell. Java Expert System Shell (Jess) is a rule engine and supports a rule-based language as well as procedural programming (Friedman-Hill, 2003). It is written in the Java language, which helps in adding rule-based technology to Java-based software systems like JADE. It is a light and fast rule engine. Jess contains a working memory, where the facts are stored. The rules in it are structured as if-then statements. The left-hand side of a rule (the *if* part or condition) is checked against the contents of the working memory of Jess. If the condition is met, then the right-hand side (the *then* part) is executed.

Jess uses a conflict strategy, which prioritizes the rules. The complexity of a rule and how long it has been in working memory are taken into account by the conflict strategy. Also, the programmer can specify the priority of a rule. The Jess rule engine uses the Rete Algorithm, which efficiently matches conditions (left-hand sides) of rules against facts in working memory. This algorithm builds a network of nodes, where each node corresponds to a pattern occurring in a rule. The Rete Algorithm consumes some additional memory for a vast increase in speed. Jess has unique features such as backwards chaining and working memory queries. It is a powerful Java scripting environment, from which the user can create Java objects, call Java methods, and implement Java interfaces without compiling any Java code (Friedman-Hill).

Jess and JADE are both Java applications. It is most common to use Jess to provide JADE agents with intelligence. The main difficulty in using Jess with a JADE agent is that a JADE agent runs on a single thread, and the rule engine may require so much time that it prevents the agent from performing its normal behavior, such as communicating with other agents. The solution is to time-limit the execution of the rule engine.

2.2 Extensible markup language

Extensible markup language (XML) is a W3C standard that facilitates the sharing of data

across the web (Singh & Huhns, 2005). It is called extensible as the tags are defined by the user. It is a well formed language. For example, the elements written do not overlap, i.e., the tags created are nested as the opening and closing tags of one element may not alternate with the tags of another element. It is much more useful to have an XML document that is valid with respect to a schema than a document that makes no reference to a schema.

Originally, XML schemas had to be expressed with DTDs (Document Type Definitions). A DTD is a document that uses an extended form of BNF to specify the abstract syntax of conforming XML documents. Today, most extensive XML schemas are written in the XML Schema Definition Language (Singh & Huhns, 2005), which is more powerful than the DTD language. For example, it allows the user to define new data types and to handle namespaces naturally. A document written in this language is itself an XML document. XML Schemas (but not DTDs) handle namespaces for elements and attributes, which allows standards to be developed modularly even when they will be applied in one and the same XML document.

2.2.1. Java architecture for XML binding. XML and Java technology are recognized as ideal building blocks for developing Web services and applications that access Web services. A new Java API, called Java Architecture for XML Binding (JAXB) (Ort), can make it easier to access XML documents from applications written in the Java programming language. Figure 4 shows the architecture of JAXB. For accessing an XML document, there are two steps: binding the schema and unmarshalling the document. The initial step is to bind the schema for the XML document, that is, to generate a set of Java classes representing the schema.

JAXB implementations are endowed with a tool named binding compiler to bind a schema (the way the binding compiler is invoked can be implementation-specific). For example, the JAXB Reference implementation provides a binding compiler that you can invoke through

scripts.

The next step performed in JAXB is unmarshalling the XML document. Unmarshalling is creating a tree of content objects that mirrors the content and layout of the document. The content tree produced from JAXB is more efficient than a DOM based tree.

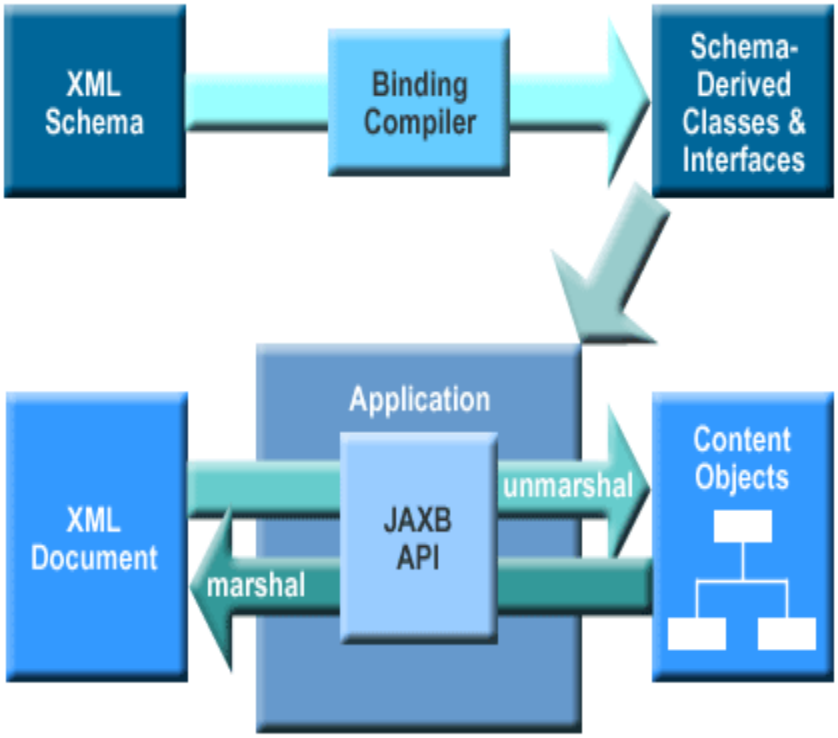

Figure 4. Architecture of JAXB (W3 Committee).

Building an XML document from Java content involves binding the schema (as before), creating the content tree, and marshaling the content tree. Binding the schema is the same process that is performed prior to unmarshalling the document. The content tree represents the content that one wants to build into the XML document. The content tree can be created by unmarshalling XML data; a class can be used that is generated by binding the appropriate schema. Marshalling is the opposite for unmarshalling. It creates an XML document from a content tree.

2.3 Web Services

A Web service is a software system designed to support interoperable machine to machine interaction over the Web (W3 Committee). These are generally based on Worldwide Web Consortium (W3C) and the Organization for the Advancement of Structured Information Standards (OASIS) standards. W3C develops open standards for interoperable technologies and its mission is to lead the Web to its full potential. OASIS aids in development, convergence, and adoption of open standards for the global information society.

The three types of participants in Web services are the service provider, the service broker and the service requester. A service provider creates the Web service and advertises it to potential users by registering the service with a service broker. A service broker maintains a registry of advertised or published services, that is, yellow pages, by which it introduces service providers to service requesters. A service requester (or consumer) searches the yellow pages for suitable service providers and then contacts a service provider to use its service(Singh & Huhns, 2005).

SOAP (Simple Object Access Protocol) is an XML-based protocol for the exchange of information in a decentralized, distributed environment. A SOAP document consists of three parts: Envelope, Schema and a convention (W3C Team). SOAP can be used in combination with several protocols, HTML being by far the most common.

Web service definition language (WSDL) is an XML language that describes the programmatic interface to a Web service. Its description includes the definition of data types, input and output message formats, the operations provided by the service, network addresses, and protocol bindings (Singh & Huhns, 2005).

Apache Axis2 (Apache eXtensible Interaction System)(The Apache Software

Foundation) is a popular, open-source SOAP engine and is the one used in this research It is a framework for developing Web service producers and consumers. It supports WSDL, allowing the user to easily build stubs, to access remote services, and to automatically export machine-readable descriptions of deployed services. Axis2 enables to easily perform the following tasks: Send and receive process SOAP messages, create a Web service out of a plain java class etc.

Universal Description, Discovery and Integration (UDDI) (UDDI Spec Technical Committee) is an OASIS yellow pages standard. It defines a standard method for publishing and discovering the network-based software components of a service-oriented architecture (SOA). When a service registers with a UDDI registry, two SOAP messages are exchanged: the first establishing authentication and the second registering the web service. The work proposed here will use Apache's jUDDI (The Apache Software Foundation), which is an open source Java-based implementation of a UDDI registry. It has been designed to run as the UDDI front-end on top of existing directories and databases. jUDDI-enabled applications can look up services in the UDDI registry and then proceed to "call" those Web services directly.

2.3.1. Web services integration gateway. Web Services Integration Gateway (WSIG) (Weiss, 1999) provides interconnectivity between the Web services domain and a JADE platform in a transparent fashion. WSIG offers bidirectional discovery and remote invocation of Web services by JADE agents and of JADE agent services by Web clients.

Figure 5 shows the architecture of WSIG. It contains the standard JADE DF, whose functionality, however, has been slightly modified to detect when an agent service registration is made by an agent that intends to publish a Web service. These requests are registered with the DF and forwarded to the WSIG Gateway Agent.

The WSIG Gateway Agent manages the entire WSIG system. The functionality of this

system includes receiving agent service registrations from the JADE DF and translating them into the corresponding WSDL descriptions, which are registered with the UDDI yellow-pages. The system functionality also includes receiving an agent service request from a Web service client, finding the service in the UDDI repository, translating the request into ACL, and sending it to the requested agent; an agent response is translated into SOAP and sent to the requesting Web service.

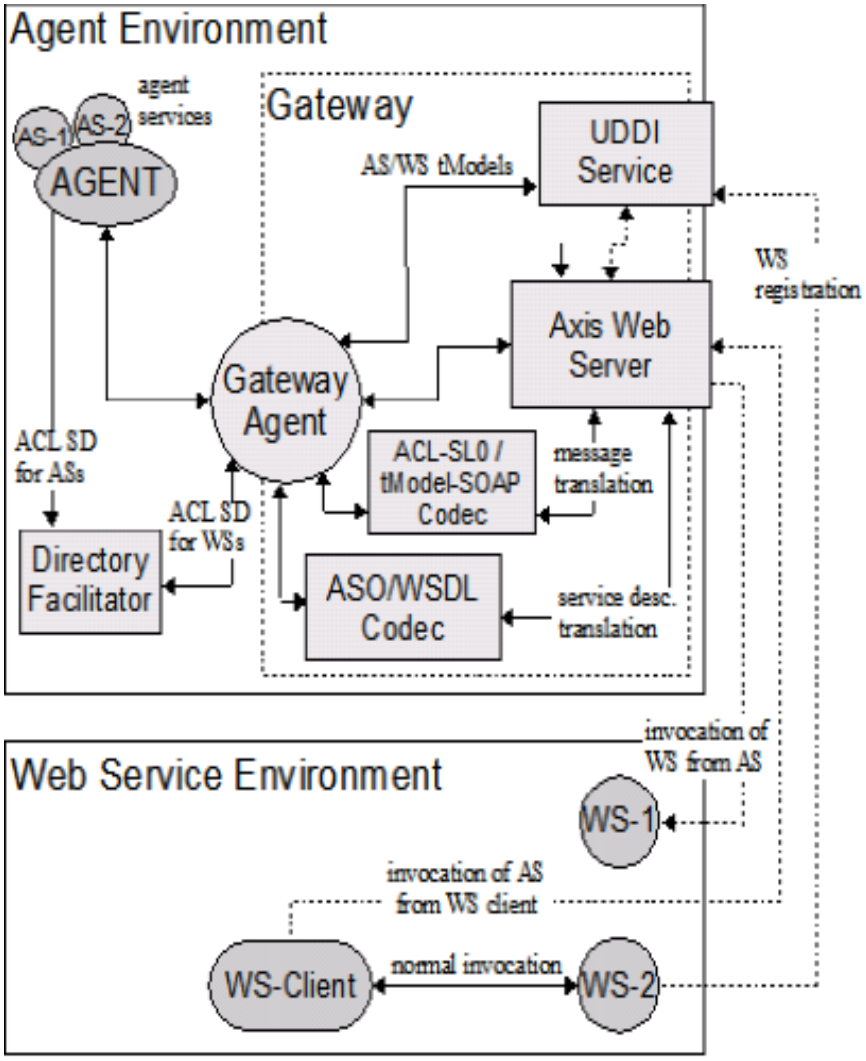

Figure 5. Web Services Integration Gateway Architecture (Greenwood, 2005).

2.4 Distributed Event based system

Event based systems consist of reactive components that cooperate by exchanging information and control in the form of events (Mühl, Fiege, & Pietzuch, 2006). Components are physically distributed over a network in distributed event based systems. The integrated components of the system communicate with each other by generating and receiving event notification. An *event* is a significant occurrence or happening. A notification is a datum that reifies an event, i.e., it contains data describing an event.

Several notifications can be fashioned to describe the same event but from multiple viewpoints. This may be done due to application or security reasons or simply because notifications are encoded in different data models. The most common data models are name/value pairs, objects, and semi structured data, e.g., XML.

Notifications are conveyed via messages. Notification or a request for information about an event can be transported by any messaging system. . A component usually generates an event notification when it wants to let the “external world” know that some relevant event occurred either in its internal state or in the state of other components with which it interacts. When an event notification is generated, it is propagated to any component that has declared interest in receiving it. The generation of the event notification and its propagation are performed asynchronously (Carzaniga & Fenkam, 2004).

Figure 6 depicts the architecture of an event based system. In this architecture, the style is characterized by a connector named *event notification service*, which is responsible for dispatching event notifications. The components are classified into *recipients* and *objects of interest*. Objects of interests are also known as producers since they publish notifications. A producer sends a notification about the happening of an event by sending a publish request to the

event service. The published notifications are not specific to the particular set of recipients. The event service responds to a publish request by forwarding it to all the subscribed recipients.

Recipients (also referred to as consumers) react to notifications delivered to them by the notification service. Consumers, like producers, are unaware of their specific communication peers. Consumers declare their interest in receiving event notifications by issuing a subscribe operation offered by the event service (Krishnamurthy, 2010). Consumers issue subscriptions to show in which event notifications they are interested and are not concerned about the producers. A component can behave as an object of interest or a recipient.

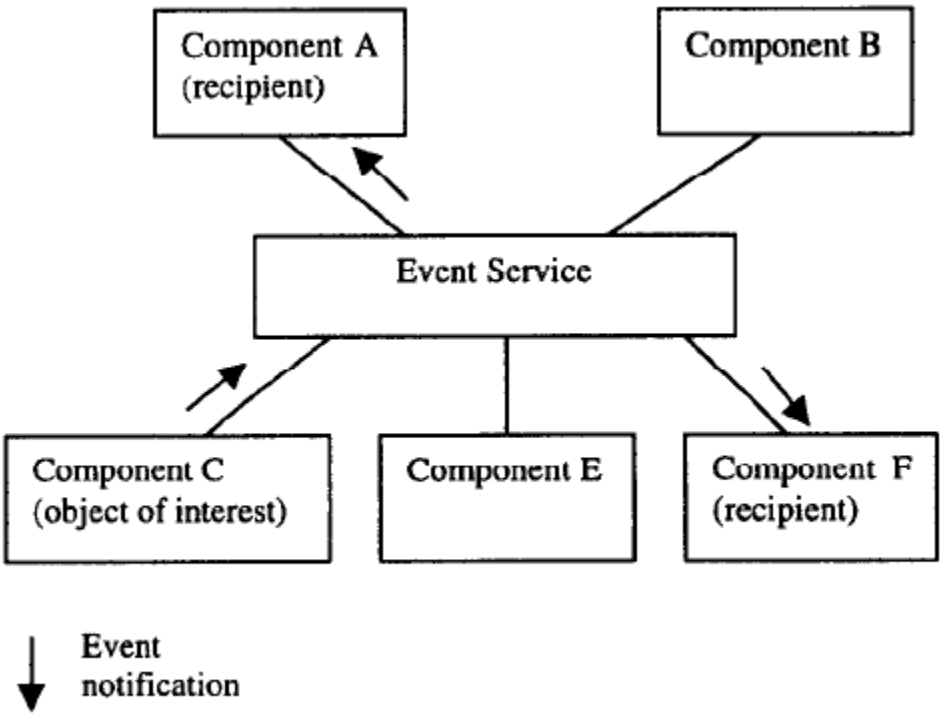

Figure 6. Architecture of an event-based system (Mühl et al., 2006).

Consumers register their interest in receiving certain kinds of notifications by submitting subscriptions to the notification service. The service evaluates the subscriptions on behalf of the consumer and delivers those notifications that match one of the consumer's subscriptions. Subscriptions are filters, which are basically Boolean-valued functions that test a single

notification and return either true or false. They can additionally include (Meta) data to govern notification selection beyond a per-notification level.

2.4.1. Java message service. Java Message Service (JMS) is an enterprise message service API specification created by Sun Microsystems (Haefel & Chappell, 2001). It is an abstraction of the classes and interfaces desired by messaging clients when communicating with messaging systems. JMS products give us a powerful means for creating enterprise messaging and they provide a secure way to transmit the information. JMS is a good tool for implementing E-commerce and enterprise applications along with other technologies like XML. JMS solutions on their own provide excellent messaging solutions but, when combined with other technologies, their capability significantly increases.

A JMS application consists of a JMS provider, JMS clients (both producers and consumers), messages and administered objects. A JMS provider is message-oriented middleware (MOM) that implements the JMS interfaces. It is mainly responsible for administrative and control features. A JMS provider is similar to an event notification service. JMS clients are programs written in Java and are responsible for producing and consuming messages. Messages are objects that pass information between JMS clients. A JMS client that produces a message is called a producer, and a JMS client that receives a message is called a consumer. Generally, a JMS client can be a producer and also a consumer. Administered objects are defined as preconfigured JMS objects that are created by an administrator and used by clients. There are two kinds of JMS administered objects: destinations and connection factories. A destination is the object a client uses to specify the target of messages it produces and the source of messages it consumes. A connection factory is the object a client uses to create a

connection to a provider. A connection factory encapsulates a set of connection configuration parameters that has been defined by an administrator.

JMS provides two types of messaging models, namely, publish/subscribe and point-to-point queuing. These messaging models are referred to as messaging domains. The names of the domains are self-descriptive; the publish/subscribe domain is a model for broadcasting messages from one sender to a set of receivers, and the point-to-point domain is a model for sending a message from one JMS client to another (Sjöberg, 2007). In the publish/subscribe messaging domain, destinations are called topics whereas, in the point-to-point messaging domain, destinations are called queues

In the publish/subscribe domain, one producer can send a message to many consumers through a topic, a virtual channel to which consumers may subscribe. Any messages addressed to a topic are delivered to all the consumers who are registered for that topic. Every consumer receives a copy of each message. The publish/subscribe messaging model is a push-based model, where messages are automatically broadcasted to consumers without them having to request the topic for new messages. The producer sending the message is independent of the consumers receiving the message.

The point-to-point messaging model allows JMS clients to send and receive messages both synchronously and asynchronously via virtual channels known as queues. A client that sends a message to a queue is called a sender, and the client receiving the messages is called the receiver. A queue may have multiple senders but can have only one receiver at a time, or, in other words, each message in the queue may only be consumed by one of the queue's receivers. The point-to-point messaging model has traditionally been a pull- or polling-based model, where messages are requested from the queue instead of being pushed to the client automatically. In

JMS, however, an option exists that allows point-to-point clients to use a push model similar to publish/subscribe. In both the domains, the use of virtual channels decouples the producers from the consumers, which allows JMS clients to be added or removed dynamically at runtime, thereby allowing the system to grow or shrink in complexity over time.

Messaging products are inherently asynchronous: There is no fundamental timing dependency between the production and the consumption of a message. The JMS server acts as an intermediary storage container for the messages that are being sent from one client to the other. However, messages can in fact be consumed both synchronously and asynchronously. In synchronous consumption, a subscriber or receiver explicitly fetches the message from the destination by calling the receive method. The receive method blocks the receiving client until the message arrives or a time-out occurs because a message does not arrive within a specified time limit.

In asynchronous consumption, a receiving client can register a message listener (similar to an event listener) with a consumer. Then, whenever a message arrives at the destination, the JMS provider delivers the message by calling the listener's `onMessage` method, which acts on the contents of the message.

JMS is not just another event service. It is designed to cover a broad range of enterprise applications, including enterprise application integration (EAI), push models, and others. JMS offers flexibility of integration by providing publish/subscribe and point-to-point functionality and allows for a robust, service-based architecture.

2.4.2. JMS-agent gateway. The JMS-Agent Gateway makes Enterprise Message Services (EMS) and their messaging models available to agent-based systems. The key role of the JMS-Agent Gateway is to provide a bridge over which non-agent EMSs are accessible to

agents in an agent-oriented fashion (E. Curry, Chambers, & Lyons, 2004). With direct access to the messaging models, an agent can easily interact with a variety of information dissemination services.

The JMS-Agent Gateway mainly consists of a JMS facilitator agent that receives agent requests for the JMS provider and correspondingly converts the request into a JMS action. The gateway is also responsible for making the connection to the JMS provider to perform subscription and message processing tasks on behalf of its clients. The JMS facilitator agent is responsible for publishing messages to JMS destinations, forwarding messages to consuming agents based on their subscription requests, and subscription administration. Publishing requests made of the facilitator by client agents follows the FIPA Request interaction protocol, and agent subscription requests follow the FIPA Subscribe interaction protocol (E. Curry et al., 2004).

Currently, the gateway supports messages of type `javax.jms.TextMessage` and `javax.jms.ObjectMessage`. The gateway design uses the JMS API; this enables any JMS compatible EMS to interact the gateway. Platform administrators are able to choose the most appropriate EMS implementation for their agent platform, and to easily change this implementation if required (E. Curry et al., 2004).

2.5 Orchestration and Workflows

The TITAN WOS (Workflow Orchestration Support) service generates and executes workflows, interacting with other agents and TBS (Titan Battle Command Support). A workflow is one way to orchestrate services. Service orchestration can be better understood by comparing it with another service composition, service choreography, and vice versa. In service choreographies, the logic of the message-based interactions among the participants is specified from a global perspective whereas in service orchestration, the logic is specified from the local

point of view of one single participant called an orchestrator (Erl, 2005). Workflows are ways to orchestrate services; Web services can be orchestrated or choreographed. Multiagent systems are typically choreographed. WADE is an extension of JADE that supports workflows and is used in the proposed work for implementing the TITAN WOS service. The remainder of this section discusses about orchestration, workflows, and WADE.

2.5.1. Orchestration. Orchestration is one way to structure concurrent activity. Concurrency has become a practical problem with the introduction of multi-core and multi-CPU systems. It became difficult, however, for programmers to manage threads and locks explicitly. These basic concurrency constructs do not serve well to express the high-level structure of control flow in concurrent programs, especially when handling failure, time-outs, and process termination (Kitchin, Quark, Cook, & Misra, 2009). Many attempts have been made to rethink the prevailing concurrency model of threads with shared state and to put forth programming languages based on novel approaches. Concurrency is also an issue in business processes, where orchestration is defined as an executable business process that interacts with internal and external services. It describes how services interact at the message level, with an explicit definition of the control and data flow (Barker & Hemert, 2008). The hub-and-spoke model is the common implementation of orchestration, which allows multiple external participants to interact with a central orchestration engine (Erl, 2005). Some of the concurrent programming languages are discussed below, followed by business process orchestration and automatic construction of orchestration models.

Orc (Kitchin et al., 2009) is a new language for distributed and concurrent programming that provides uniform access to computational services, including distributed communication and data manipulation, through *sites*. By means of four simple concurrency primitives, the

programmer *orchestrates* the invocation of sites to achieve a goal while managing timeouts, priorities, and failures. Orc was first developed as a process calculus (Misra 2004) designed to express orchestrations and wide-area computations in a simple way. Its purpose was to overcome the problem of concurrency in multi-core/CPU systems. It is inherently concurrent and implicitly distributed and has a clearly specified operational semantics.

Erlang is a concurrent programming language, originally designed by Ericsson (Vermeersch, 2009). (Erlang is named after A. K. Erlang, a Danish mathematician, but "Erlang" is also an abbreviation of "Ericsson Language.") It is designed for programming concurrent, real-time, distributed fault-tolerant systems and is based on the actor model of concurrency (Armstrong, Virding, Wikström, & Williams, 1996). It is often compared with the Orc language. The Actor model was developed to overcome problems in concurrency arising from threads and locks. In the Actor model, every object is viewed as an actor (Hewitt, Bishop, & Steiger, 1973). An actor contains a mailbox and a behavior. Messages exchanged between actors are buffered in the mailbox. After an actor receives a message, its behavior is executed, and the actor can send any number of messages to other actors. It can also create any number of other actors. In this model, all communication is performed asynchronously and all communication is through messages (Vermeersch, 2009). One more important property of this model is locality. Locality obtains when an actor processing a message can send messages only to an actor whose address is given in that message, whose address it has before it received the message, or that was created while processing the message.

Hewitt, who developed the Actor model, points out (Hewitt, 2007) that hardware development is expanding in local and nonlocal massive concurrency. Local concurrency is used by new 64-bit multi-core microprocessors and multi-chip modules whereas nonlocal concurrency

is used for wired and wireless broadband packet switched communications. Hewitt claims that all these developments favor the Actor model and that actors reach the level of agents when they can express notions like those of intention, plan, contract, belief, and policy,

Oz (Haridi & Franzen) is a multi-paradigm language designed for advanced, concurrent, networked, soft real-time and reactive applications. It has features of object-oriented programming and functional programming. Any number of sequential threads can be created dynamically by users.

Pict (Pierce & Turner, 2000) is a concurrent functional language based on the π-calculus and is similar to Orc language, which is based on the Orc calculus. The main difference between Pict and Orc is that Pict imposes a functional, continuation-passing structure on an unstructured communication calculus whereas the Orc calculus is already structured.

The Business Process Execution Language (BPEL) (OASIS WSBPEL Technical Committee, 2007) is a de-facto approach for orchestrating Web services and is a standard for executing processes. BPEL is supported by companies such as ORACLE, IBM and Microsoft. It supports two types of business processes, abstract and executable. An abstract business process is a partially specified process that does not reveal the internal behavior of the system and is not meant to be executed. An executable process is a completely specified process that is ready to execute. A BPEL process itself is a kind of flow-chart, where each element in the process is called an *activity* (Wil M.P. Van der Aalst et al., 2005).

Business process mining, or process mining for short, supports extracting information from event logs and is closely related to data/workflow mining. For example, the audit trails of a workflow management system or the transaction logs of an enterprise resource planning system can be used to discover models describing processes, organizations, and products (Process

Mining Board). Process mining aims at the automatic construction of models explaining the behavior observed in the event log. For example, based on some event log, one can construct a process model expressed in terms of a Petri net (W.M.P Van der Aalst et al., 2007). ProM(process Mining Board) is a generic open-source framework for implementing process mining tools in a standard environment.

2.5.2. Workflows. A workflow is a process (W. M. P. Van der Aalst & Hee, 2002). A process is described as a collection of tasks, conditions, and sub-processes. A process is basically designed to deal with a particular category of cases and to handle many individual cases. A process can consist of a number of tasks that need to be carried out and a set of conditions that determine the order of the tasks. A task is a logical unit of work that is carried out as a single whole resource. A resource is the generic name for a person, machine, or group of persons or machines that can perform specific tasks. A process is also sometimes called a procedure.

A workflow is a process, and each process has tasks to perform. A task can be optional, which means that it needs to be carried out for only certain cases. Which optional tasks are performed may depend on the order in which tasks are performed. There are four constructions for routing, namely, sequential routing, selective routing, parallel routing, and iteration. With sequential routing, tasks are carried out one after another. Selective routing is when there is a choice between two or more tasks; the choice depends upon the specific properties of the case. Parallel routing is when several tasks can be carried out at the same time or in any order. In iteration, a task is performed repeatedly until some condition holds.

The elementary technique for analyzing a workflow is reachability analysis, which starts with a Petri net that specifies the possible behaviors of the modeled system. It is assumed that a Petri net representing a workflow process, that is, a *workflow net*, has a unique entrance place,

start, and a unique exit place, *end*. A reachability graph indicates possible transitions between Petri net states, starting with the initial state. In a Petri net with *n* places, a state is an *n*-tuple giving the number of tokens in each place. A state is reachable if there is a directed path in the reachability graph from the initial state to it. Van der Aalst defines a *soundness* property, a minimum requirement that every process must meet, as follows: A process contains no unnecessary tasks and every case submitted to the process must be completed in full and with no references to it (that is, case tokens) remaining in the process. More explicitly, we define a workflow net to be sound if it meets the following requirements:

- For each token put in the place *start*, one (and only one) token eventually appears in the place *end;*
- When the token appears in the place end, all the other places are empty; and
- For each transition (task), it is possible to move from the initial state to a state in which that transition is enabled.

This definition of soundness assumes a notion of *fairness*: if a task can be executed, then its execution cannot be postponed indefinitely. The soundness of the workflow net corresponds to two standard properties—*liveness* and *boundedness*—of the so-called short-circuit version of the Petri net, where a transition is added that has *end* as its sole input place and *start* as its sole output place. (Hence it "re-circulates" tokens). A Petri net is live if, from every reachable state, every transition can eventually be fired. A Petri net is bounded if there is a limit to the number of tokens in each place when the process is started in the initial state. A workflow net, then, is sound if and only if it is short-circuit version is live and bounded.

Based on an analysis of existing workflow management systems and workflow languages, a new workflow language called Yet Another Workflow Language (YAWL) was

developed by van der Aalst and ter Hofstede in 2002. YAWL (YAWL Team) is a Business Process Management (BPM)/Workflow system. It handles complex data, transformations, integration with organizational resources and Web Service integration. Petri nets are used as a base, and mechanisms are added that allow more direct and intuitive support of the workflow patterns (W. M. P. Van der Aalst & ter Hofstede, 2005).

Scientific workflows have been developed by creating workflows for automating large-scale science (Deelman & Gill, 2006). A scientific workflow captures a series of systematic steps that describes the design process of computational experiments. Scientific workflow systems provide an environment to aid the scientific discovery process through the combination of scientific data management, analysis, simulation, and visualization (Barker & Hemert, 2008).

2.5.3. Workflow and agents development environment. Workflow and Agents Development Environment (WADE) adds to JADE functionality for the execution of the tasks defined according to the workflow metaphor and a number of management mechanisms (WADE Board). WOLF, an eclipse plug-in for WADE, facilitates the creation of WADE-based applications. WOLF provides a graphical form for depicting a process in WADE (Figure 7).

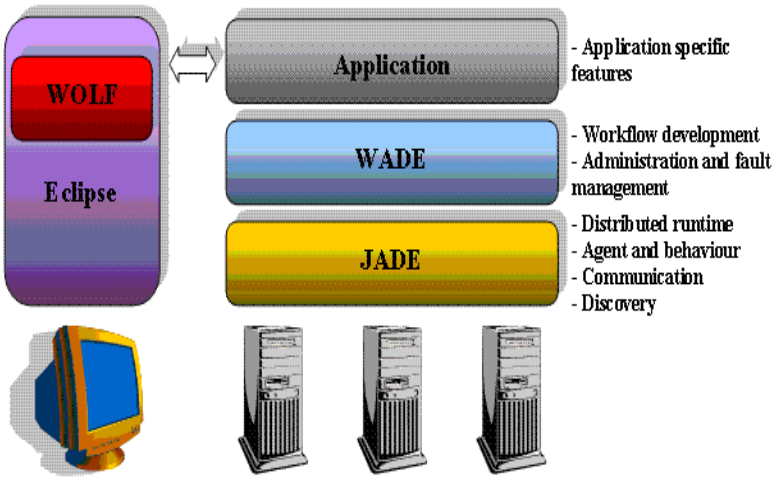

Figure 7. WADE Platform (WADE Board).

The most important aspect of WADE is to bring the workflow approach from the business process level to the level of internal system logic. WADE does not target the orchestration of the high level services provided by different systems but rather the implementation of the internal behavior of each single system. So WADE should be particularly suitable for addressing the execution of possibly long and fairly complex tasks, while state of the art middleware technologies are more appropriate with systems that mainly manipulate data in data repositories following user requests.

WADE adds to JADE the following features.

- The capability to define agent tasks according to the workflow metaphor.
- An architecture, additional components and mechanisms that facilitate the administration of a distributed WADE-based application in terms of configuration, activation, and monitoring.

These two features can be used separately, so WADE-based applications can be classified under two main categories: simple JADE applications, where some agent tasks are defined as workflows, and applications structured according to the complete WADE architecture, which are thus able to exploit all WADE features.

WADE workflows are composed of different activities as calling a Web Service, invoking another workflow or executing raw code directly included in the workflow class. Different set of activities in WADE are *Tool* activities, *Code* activities, *Web services* activities, *Subflow* activities, *Subflow* join, *Route* activities.

2.6 Conceptual Background

This section introduces two main concepts for coordination between TBSs of TITAN. First sub section discusses about common knowledge and next discusses about theory of mind.

2.6.1. Common knowledge. Common Knowledge is a prerequisite for coordination among human or artificial agents. Since most of the environments we face are dynamic, the common knowledge agents initially have generally does not suffice for agents to coordinate their actions in each and every situation. So attaining common knowledge is a serious issue (Wiriyacoonkasem & Esterline, 2001).

Common knowledge has been studied in many disciplines such as philosophy, linguistics, game theory, and distributed systems. It is a necessary condition for Common Knowledge. To explain this concept (Fagin, Halpern, Moses, & Vardi, 1995), let G be a group of n agents, denoted by ordinals, G = (1, 2… n). We introduce n modal operators K_i, $1 \leq i \leq n$, where $K_i\ \varphi$ is read "agent i knows thatφ". $E_G\varphi$, read as "everyone (in G) knows that φ," is defined as $K_1\varphi \wedge K_2\varphi \wedge \ldots \wedge K_n\varphi$. Let E_G^k be the E_G operator iterated k times. Then "it is common knowledge in group G that φ," in symbols, $C_G\ \varphi$, is defined with the infinite conjunction $E_G^1\ \varphi \wedge E_G^2\ \varphi \wedge \ldots \wedge E_G^i\ \varphi \wedge \ldots$". For an intuitive example, note that traffic lights would not work unless it were common knowledge that green means go, red means stop, and that lights for opposite directions have different colors. If this were not so, we would not confidently drive through a green light. In a standard epistemic logic (i.e., logic of knowledge) augmented with the operators defined above, it is easy to show that, if everyone in G agrees that ψ, then the agreement is common knowledge (Fagin et al., 1995). It can also be shown formally that coordination implies common knowledge.

Besides characterizing common knowledge in terms an infinite conjunction, Barwise (1989)identifies two other approaches. In the fixed-point approach (which eliminates the infinite conjunction), we view $C_G\ \varphi$ as a fixed-point of the function(Fagin et al., 1995) $f(x) = E_G(\varphi \wedge x)$. The third approach (which we formulate following (H. H. Clark & Carlson, 1982)) is the shared

situation approach. For this, where A and B are rational, we may infer common knowledge among A and B that φ if

- A and B know that some situation σ holds.
- σ indicates to both A and B that both A and B know that σ holds.
- σ indicates to both A and B that φ.

This is termed the Mutual Belief Induction Schema; we take "mutual belief" and "common knowledge" to be synonyms. Barwise concludes that the fixed-point approach (essentially implied by the shared situation approach) is the correct analysis of common knowledge, and that common knowledge generally arises via shared situations. H. H. Clark and Carlson (1982) identified three "co-presence heuristics" giving rise to different kinds of shared "situations." Two, physical co-presence (e.g., shared attention) and linguistic co-presence (as in conversation), properly relate to situations, but the third, community membership (presupposed by the others), is not temporally or spatially restricted. It is essentially the social part of what A. Clark (1998) called *scaffolding*: a world of physical and social structures on which the coherence and analytic power of human activity depends. Let "common state knowledge", abbreviated "CSK", refer to what is established by the two non-scaffolding heuristics. Common knowledge thus is either scaffolding or a self-referential feature of the situation.

2.6.2. Theory of mind. In the intended use of TITAN, each unit has its own copy of the TITAN suite of services, called a TBS (Titan Battle Command Support). Since there is a hierarchy of commanders, there needs to be a hierarchically integrated set of TBSs, where a parent in the hierarchy coordinates its children. Coordination of child TBSs by a parent TBS using the same coordination paradigm as used within a TBS—a multiagent system—is infeasible because of the communication requirements. So inter-TBS communication uses JMS. In Chapter

4, it will be proposed that the appropriate TITAN service in a TBS maintain a WADE workflow involving agents that are proxies for its child TBS. This workflow will determine what JMS messages are published for the child TBS, and JMS messages published by child TBS will result in updates to the proxy agents maintained by the parent TBS. The claim here is that use of proxy agents in this way implements a scaled-back version of what is known as theory of mind (ToM) in developmental psychology (Flavell, 2004). Also, ToM (it is claimed) provides an essential means for a group to achieve CSK (Esterline, Wright, & Banda, 2011).

Theory of mind is the ability to attribute mental states (beliefs, desires, intentions, assumed roles, etc.) to one and others and to understand that others have mental states different from one's own. Early work in Theory of mind (ToM) focused on the striking improvement in children between ages three and five on false-belief (FB) and Level 2 visual perspective-taking (PT) tasks. As an example of an FB task, the child watches as a puppet sees a cookie put in one of two boxes and leaves. In the puppet’s absence, someone moves the cookie to the other box. When the puppet comes back, the older child (having a notion of false belief) says the puppet will look in the original box. The younger child (not having this notion) will say the puppet will look in the box with the cookie. As an example of a TB task, the older child, but not the younger, understands that a picture book oriented correctly for them on the table looks upside down to a person seated opposite. ToM development provides a scaffold for early language development: when a child hears an adult speak a word, they recognize that the word refers to what the adult is looking at.

Wiriyacoonkasem and Esterline (2001) have shown how the physical co-presence heuristic may be used for groups of artificial agents to attain common knowledge by perceptual means. To focus on the episodic nature of co-presence evidence, a modal operator *S* is

introduced into epistemic logic: $S_a^t\ \varphi$ indicates that agent *a* sees at time *t* that φ. Given time parameters for other operators, they introduce the axioms $S_a^t\ \varphi \Rightarrow K_a^t\ \varphi$ and (for simplicity) $K_a^t\ \varphi \Rightarrow K_a^u\ \varphi$ for all $u > t$ and focus on cases where, e.g., both $S_a^t\ S_b^t\ \varphi$ and $S_b^t\ S_a^t\ \varphi$ hold. The formalism exposes strong reasons to hold that, to attain CSK, agents must model each other's perceptual abilities, which requires common knowledge of shared abilities. The linguistic co-presence heuristic can in principle be handled somewhat similarly, but hearing what another says involves linguistic conventions and apparently relies on scaffolding that develops as part of our biological endowment; in addition, what is conveyed is not immediately veridical. In any case, the general point is that, with the appropriate scaffolding, ToM abilities provide a mechanism for achieving CSK.

CHAPTER 3

Tactical Information Technologies for Assured Net Operations

An Advanced Technology Objective (ATO) funded by the Army is The Tactical Information Technologies for Assured Net Operations (TITAN) Program. The TITAN description presented here, unless otherwise attributed, is based on *TITAN ID&M: A Guide for Developing Agent-Based Services*(US Army, 2008), which is the definitive document on the TITAN Program The primary and main goal of this ATO is to demonstrate how emerging information technologies can significantly improve key areas of tactical operations resulting in the transition of software developed under the ATO to existing battlefield systems.

The key areas of tactical operations that are the principal targets for improvement through the TITAN ATO are:

- Information Dissemination and Management (ID&M)
- Network Management (NM)
- Information Assurance (IA)

The NM sub-project addresses concepts related to network reliability and bandwidth capacity, which are significant for timely dissemination of data. The IA sub-project addresses the problem of security and sensitivity of tactical information that is to be disseminated.

The TITAN ID&M sub-project addresses the problem of different systems using different internal data representations by developing an integrated set of *Battle Command Services* that project a common information exchange data model compatible with the *Joint Consultation, Command and Control Information Exchange Data Model* (JC3IEDM) (Data Management Working Group, December 13, 2007), which is a model that, when physically implemented, aims to enable the interoperability of systems and projects required to share command-and-control

information (Headquarters, August 11, 2003). These Battle Command Services will satisfy the need to create *operational orders* (OPORDs) as output from mission planning activities as well as to update them (by producing *fragmentary orders*, referred to as FRAGOs) during mission execution. OPORDs from the commanders are used to synchronize military operations, and an OPORD contains the descriptions of the following:

- Task organization.
- Situation.
- Mission.
- Execution.
- Administrative and logistic support.
- Command and signal for the specified operation.

OPORDs are a particularly important kind of command and control (C2) product or information type (Headquarters, 31 May, 1997). OPORDs are directives that are issued by a commander to his/her subordinate commanders to coordinate the execution of an operation.

Another C2 information type is the operational plan (OPLAN), which is a proposal for executing a command to conduct military operations. By issuing an OPLAN, commanders may initiate preparation for possible operations. The other two types of C2 products are *fragmentary orders* (FRAGOs) and *warning orders* (WARNOs). A FRAGO gives timely changes to existing orders to subordinate and supporting commanders while providing notification to higher and adjacent commands. It mainly addresses those parts of the original OPORD that have changed. A WARNO is an initial notice of an order or an action that is to follow. WARNOs help subordinate units and their members to prepare for new missions.

An XML schema is used in the TITAN Program to represent different types of C2

products. The root XML element of every C2 product document is `prdC2`. The type attribute of a `prdC2` element specifies the type of C2 product, and the `dsg` attribute specifies the designator (authoritative source) of the C2 product (Headquarters, August 11, 2003).Commanders cannot exercise C2 alone except in the simplest and smallest organizations (Headquarters, August 11, 2003). Even at the lowest levels, a commander needs support to exercise C2 effectively.

There is a hierarchically integrated set of the C2 systems employed, as shown in Figure 8. The Information Management component of each of these systems supports the capability to disseminate information between C2 systems operating in different echelons (levels of command), as shown by the green arrows in the figure. There is exactly one active instance of an OPORD as well as a collection of other related artifacts (e.g., FRAGOs and WARNOs) associated with each instance of a C2 system.

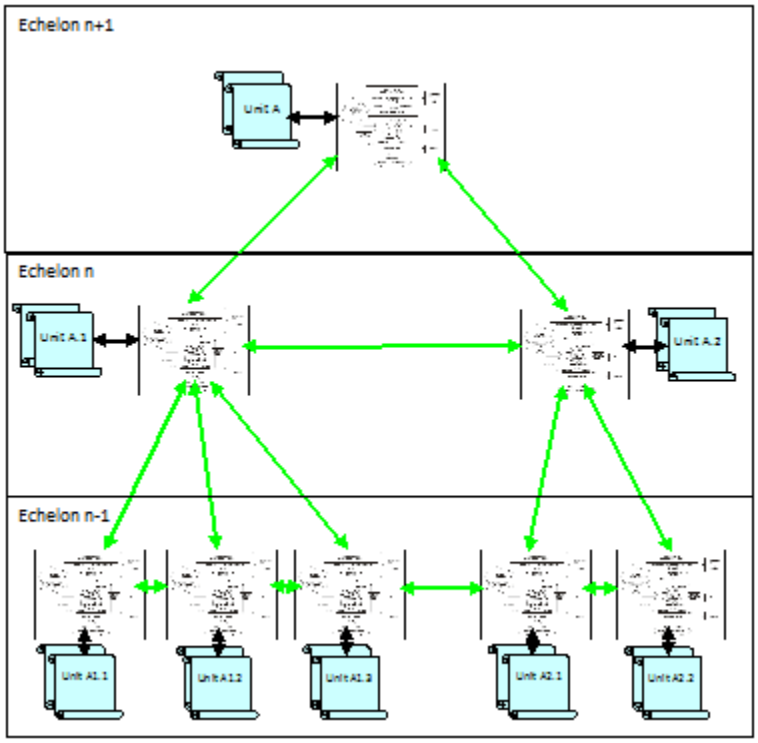

Figure 8. Information Dissemination in a Hierarchical Set of the C2 Systems (US Army, 2008).

The major types of activities being performed in the C2 system are:

- Planning
- Preparation
- Execution
- Assessment

For any OPORD, planning is followed by preparation, which is followed by execution; assessment can happen at any point. In addition, a given C2 system may be addressing several OPORDs at once, but it will be performing different activities with different OPORDs.

A TITAN Service is referred to as *a Titan Battle Command Support* (TBS). A TBS is a special type of *federate*. A federate is defined as an executing software program that knows other federates and is able to interact with them via a common communication infrastructure, which in the case of TITAN is the Java Messaging Service (JMS) (Haefel & Chappell, 2001). Since there is a hierarchy of commanders, there needs to be a hierarchically integrated set of TITAN suites. Each TBS implementation provides up to four different types of interfaces:

- a Web Service interface,
- A Data Dissemination Service (DDS) interface,
- Java Message Service (JMS) topics, and
- An interface made available through an agent framework.

The DDS interface complies with the Battle Command standards for the Army's SOA Foundation (SOAF-A) and the Battle Command standards for security when publishing information for consumption outside of the tactical operations center (TOC).

Each TBS needs to use several JMS interfaces. These include JMS interfaces to other TBSs and to the Battle Command War-fighter (BCW), a user interface that either displays information to, or receives manual input from, a human consumer of the TBS capabilities. The

other JMS interfaces are to the Common Domain Manager (CDM) and the Battle Command Query Service (BQS). In addition, each TBS must provide a JMS interface for use by other TBSs, callback type interfaces for acquiring source data using publish/subscribe, and perhaps WSDL interfaces.

The following services are planned for each TITAN suite under the ID&M part of the TITAN Program:

- the Alert and Warning Service (AWS) (CERDEC US Army, November 12, 2008a), which generates warnings, alerts, and predictions,
- the Battle Book Support Service (BBS) (CERDEC US Army, November 3, 2008), which provides decision support tools for analysis,
- the OPORD Support Service (OPS) (CERDEC US Army, December 22, 2008), which provides support services to manage C2 products like the OPORD and OPLAN,
- the Smart Filtering Support (SFS) Service (CERDEC US Army, November 12, 2008b), which enables extraction of structured data from free text reports and messages,
- the Workflow Orchestration Support (WOS) Service (CERDEC US Army, October 21, 2008), which organizes the workflow of services across the enterprise for synchronization,
- the Initialization and Continuing Operations Support Service (ICS), which provides initialization and co-ordination service, and
- The Product Dissemination Support Service (PDS), which determines the addressees for dissemination of OPORDs and associated reports.

Each of these services is stated as a TITAN service. Figure 9 shows the coordination and working of all the services listed above on a single instance of an OPORD within a TITAN suite.

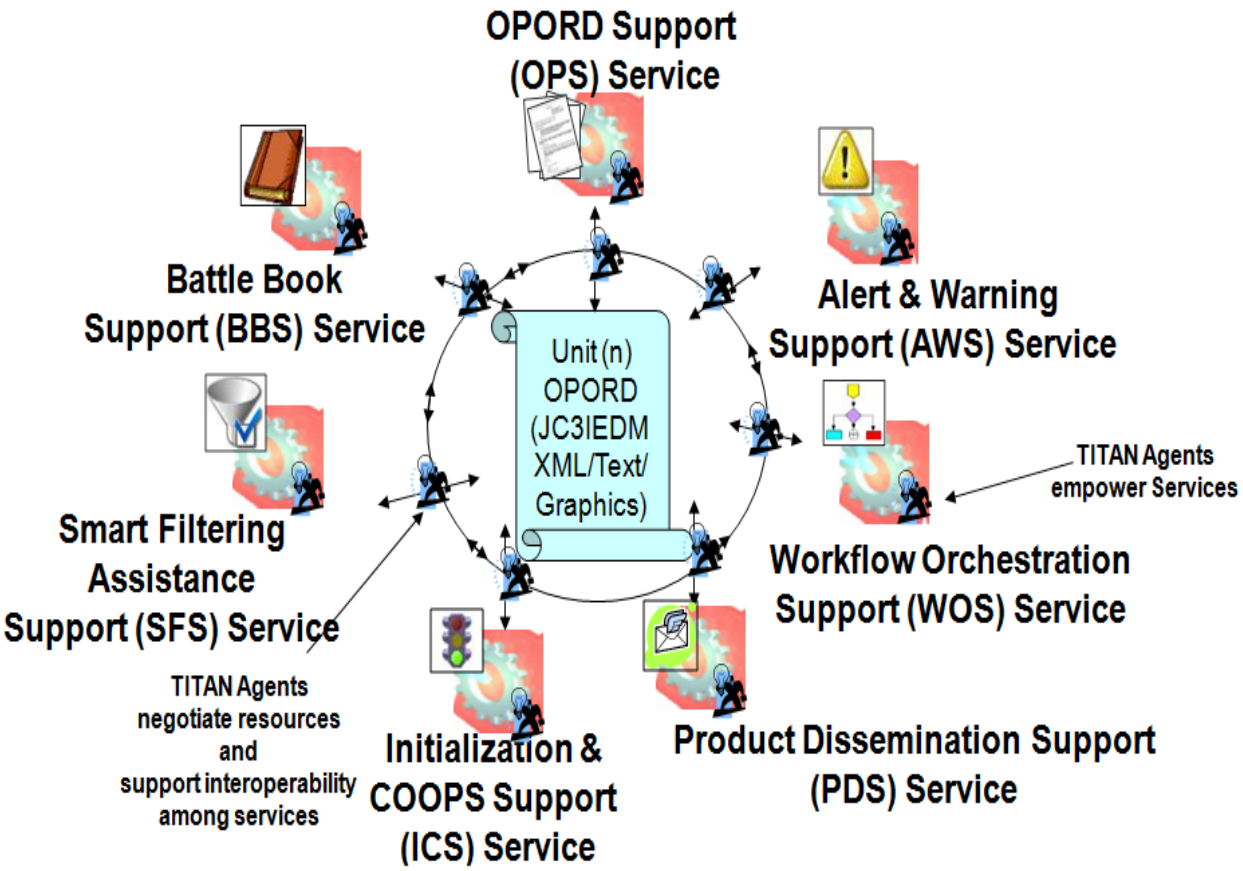

Figure 9. TITAN Services Planned in a TBS (US Army, 2008).

Each TITAN service session implements the state model shown in Figure 10. The state transitions show the relationships between the TITAN service session states and the C2 system activities. The *collaborate* state supports the planning activity, and the *execute* state supports the preparation and execution activities. Further, while in the collaborate state, the service may be supporting assessment for the purposes of planning, and, while in the execute state, it may be supporting assessment for preparing or executing. When in the *initialize* state, the service session is not ready to support the C2 system activities but initializes its own internal data structures and

communicates with other systems in preparation for supporting C2 activities. Assessment is performed in all states except the initialize state.

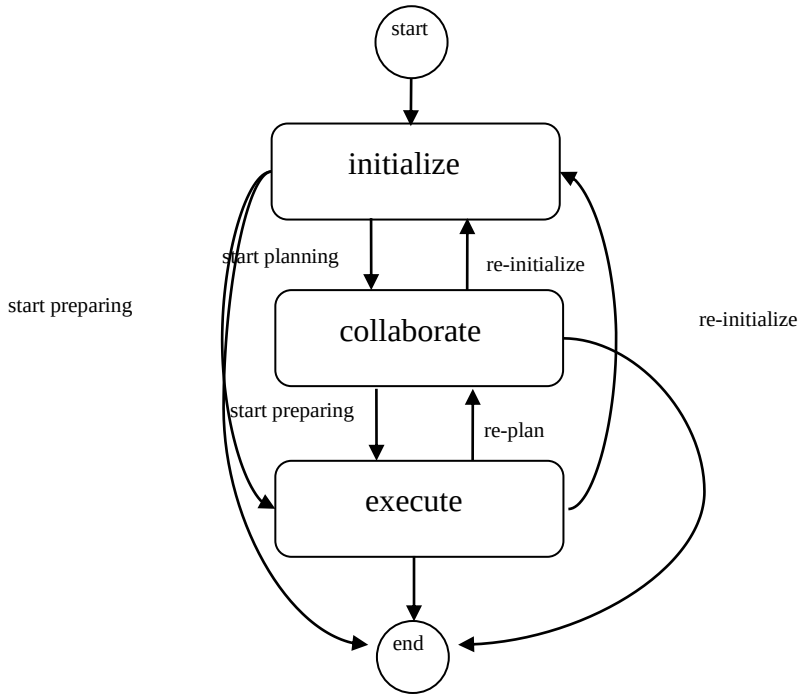

Figure 10. State Model of a TITAN Service Session (US Army, 2008).

3.1 The WOS Service

Workflow Orchestration Support (WOS) Service (CERDEC US Army, October 21, 2008) generates and executes workflows from tasks and other trigger objects, and archives messages to support status requests and situational reasoning. The gateway agents that support communication outside the service are, the SOAP Gateway Agent which provides a SOAP service accessible by clients external to the TITAN TBS community, and the JMS Gateway Agent which provides bi-directional communication between the WOS agents and other TITAN TBS services.

WOS has three collaborative agents—the Orchestration Agent, the Workflow Agent, and the Monitor Agent excluding the two gateway agents mentioned above. The Orchestration Agent

receives tasks, other trigger objects and generates workflows according to the business roles for that objects. The Workflow Agent executes workflows, interacting with other agents and TBS services. The Monitor Agent receives all incoming messages from the two gateway agents and routes outgoing messages to the correct gateway agent, and also maintains an archive of message versions to support status requests and situational reasoning. The SOAP Gateway Agent provides a SOAP service accessible by clients external to the TITAN TBS community and registers these services with a specified UDDI registry. When a SOAP call is invoked by an external client, this agent generates tasks to trigger the WOS agents to fulfill the SOAP request.

3.2 The AWS Service

According to the *Guide* (US Army, 2008), the AWS is implemented as a single *Dispatcher (Coordination)* agent, which provides coordination and monitoring of roles, and a suite of assessor agents, each handling a single instance of a Commander's Critical Information Requirements (CCIR) of a specific type. The following are the types of assessor agents (CERDEC US Army, November 12, 2008a).

- The *Area Protection Assessor* agent evaluates potential threats to a polygonal area defined in the CCIR.
- The *Route Protection Assessor* agent evaluates potential threats to a route defined in the CCIR as a sequence of line segments.
- The *Event Cluster Assessor* agent identifies clusters of Improvised Explosive Devices (IEDs) and generates an Alert Report of the appropriate severity if the number of IEDs in any cluster exceeds a CCIR-defined threshold.

The implemented prototype of the Dispatcher agent provides alerts and warnings as it monitors a very simple simulation of the execution of the OPORD. The Dispatcher agent

provides a user interface to simulate the execution of the tasks mentioned in the OPORD. The user indicates whether an alert report should be generated by the dispatcher agent through this interface. The Dispatcher agent communicates by receiving ACL messages from an assessor agent to determine the alert level.

3.3 The OPS Service

For the OPS Service (CERDEC US Army, December 22, 2008), the *Guide* specifies two Cougaar agents: the *Message* agent, handling JMS messaging in the system, and the *Update* agent, performing all operations on OPORDs and OPLANs. The *Guide*‘s description of these agents requires *blackboards* and *plugins*, which are Cougaar constructs. A plugin is analogous to a JADE agent behavior, and a blackboard is a shared memory that is used for plugin communication within and between agents.

The Message agent holds *JMS* plugin in the system and takes JMS messages off the various JMS topics and puts them on the blackboard. These messages are encoded with CERDEC’s *prdC2* schema. On startup, this agent receives OPLANs and OPORDs via the TITAN control JMS topic. During collaboration and execution, it receives reports, WARNOs, FRAGOs, OPLANS and OPORDS. From its blackboard, it relays messages it receives onto the Update agent’s blackboard for processing. It also publishes from its blackboard the OPLAN and OPORD objects relayed to it from the Update agent’s blackboard that are relevant to the unit corresponding to the current OPS instance.

The Update agent consists of the *Versioning* plugin, which subscribes to the blackboard for all messages that the Message agent JMS plugin publishes to the blackboard. The Versioning plugin republishes each such message; depending on the message’s type, various other plugins will operate on it. It also replaces older OPLAN, OPORD, WARNO and FRAGOs with newer

ones for each friendly unit indicated by the *dsg* (designation) attribute of the message's *prdC2* element. Besides the Versioning plugin, the Update agent contains the following plugins.

- The *Assess* plugin determines whether information in a report is in the polygonal area of responsibility (AoR) of the unit corresponding to the current OPS instance.
- The *Plan* plugin updates OPORDs and OPLANs based on the reports, WARNOs, and FRAGOs the Update agent receives.
- The *AoR/CM* plugin creates the AoR from topic operations (tpcOps, giving the boundary-line information) boundaries and calculates the center of mass (CM) of the AoR. It adds both the AoR and the CM to the topic mission (tpcMsn, which includes the AO) when an OPORD or OPLAN is received for the unit corresponding to the current OPS instance.

An efficient way to incorporate the features of a blackboard in a JADE agent was required, as the concept of a blackboard is not implemented in the JADE framework. The issue was resolved by providing the functionality of the blackboard using the features of a JMS topic and Java objects. Several Java objects were created from the XML document using JAXB.

CHAPTER 4

Implementation

This chapter discusses the implementation of the architecture described in the first section. The programming language used for implementation is Java and the version of JADE used here (viz., 3.6) requires JDK (Java Development Kit) 1.6 and JRE (Java Runtime Environment) 6. Other applications used to run TITAN services include JBoss Application server version 4.2.2 GA to furnish the JMS providers and webservers. WADE version 3.0 is used for executing workflows. JMS-Agent Gateway version 0.5 and WSIG add-on version 2.0 are used to allow interoperation among these technologies. The implementation also uses as the UDDI repository jUDDI version 0.9rc4 with MySQL version 5.1.34, which acts as the database for the repository. The second section of this chapter is an overview of TITAN services. The next section presents the work done by the previous MS student who worked on this project (Krishnamurthy, 2010). The remaining sections except for the last discuss the present work done. The final section describes configuration issues confronted in the previous and current phases of this effort.

4.1 Architecture

This section describes the architecture of prototype TITAN services. It mentions various services planned under TITAN. The principal interest is with three TITAN services, namely, the Alert and Warning Service (AWS), the OPORD Support (OPS) Service and Workflow Orchestration Support (WOS) Service.

The work proposed for the AWS will build on the work done by previous MS students Barnette (2008); Mason (2008); Reid (2007). Emphasis will be on coordinated use of JADE agents, JMS, and Web service. The Dispatcher agent will provide alerts and warnings as it

monitors a simple simulation of the execution of the OPORD, and it will provide the user interface mentioned above (Section 3.2). The proposed work will create only stubs for these three kinds of agents, which will communicate with the Dispatcher agent with ACL messages. Enough functionality will be provided so that reports in the appropriate format may be communicated and so that a convincing variety of behaviors may be generated for testing.

The proposed work for OPS will implement the essentials of the Message agent using the JMS Agent Gateway. The JADE blackboard implemented by the previous MS student will be used where a blackboard is specified. The proposed work will implement enough of the Update agent so that simple changes may be made to C2 products as new reports are received. This will involve implementing the critical aspects of the Versioning plugin's functionality but only enough of the functionality of the other plugins as required for communication and testing.

In the WOS Service, WADE will be used to implement the Workflow Agent. The implementations of the Monitor Agent will be not much more than a stub allowing communication with other TBSs. In the design, the Orchestration Agent generates WADE workflow code as per the OPORD, and this code is compiled by the Workflow agent before the workflow is initiated.

In the simpler design, which will actually be followed, the Orchestration Agent will simply select from a set of predefined workflows as per the information in the OPORD document. TBSs of peers and children and the agents making them up will not be directly included in the workflow of a given TBS because a unit of significant size would require a prohibitively large and involved workflow. Also, the communication between TBSs is done with JMS. So a proxy agent is introduced for each peer and child TBS. We generally expect each proxy agent to include its own subflow. Such a subflow could involve further proxy agents for

subunits two echelons down, but we stop one echelon down as direct communication stops one echelon down.

The workflow just described is provided by the Workflow Agent. The Monitor agent will be an interface for all communication into and out of the WOS. Much of this communication will be with the other TBSs, but it will also include reports from the AWS in the current TBS and updates to the OPORD from the OPS. The original OPORD will be provided directly to the Orchestration agent. A rudimentary history of the unit's message activity will be maintained by the Monitor agent. The Orchestration and Workflow agents can query this history when a decision requires past values. The history can also be queried from outside the WOS. The history will be critical for the Orchestration and Workflow agents when a work flow is aborted since they will be able to tell where the aborted workflow left off.

The Monitor agent will get input from the current unit's peers and children. Some of these messages will be for controlling the workflow. But some will represent inter-unit communication that does not directly impact the workflow. These get translated into ACL messages (with the appropriate performative) that get sent by the proxy representing the source unit to the proxy representing the target unit. At any time, a work flow and its subflows can be aborted.

We use a simplified schema for the OPORDs in which the information relates mainly to the child units and the sequence of activities (expressed with do elements) they perform. The Orchestration Agent uses JAXB to access the information in the OPORD. We assume that the OPORD defines phases and that each subunit executes a given sequence of activities during each phase.

For each subunit, the Orchestration Agent produces a Java code file defining the class for

the proxy agent for that subunit. The behavior for the class will be a workflow in which the activities for each phase will be executed in sequence. The Orchestration Agent also produces a Java code file for the class for the main workflow agent (for the unit). The behavior for this class is a workflow that has the behaviors of the proxy agents as subflows. After the Orchestration Agent produces these Java files, it sends a message to the Workflow Agent with the names of files just produced. The Workflow Agent then compiles the files and produces agents as instances of the classes. Figure 11 shows example of two sub units.

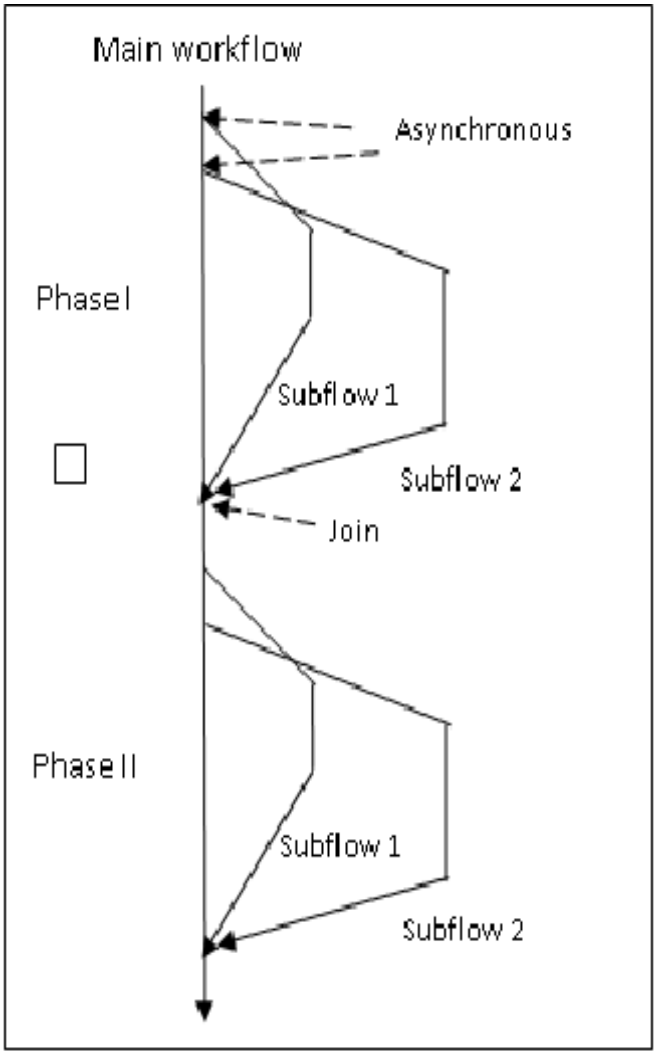

Figure 11. A Two-phase Workflow for a Unit with Two Subunits

The unit workflow forks asynchronous executions of the subflows at the beginning of each phase and joins them at the end of the phase; this is how the activities of subunits are synchronized.

Communication is facilitated by the fact that all agents initially register with the DF. The main workflow (unit) agent and the subflow (subordinate unit) agents can communicate directly with the Monitor Agent and thus with the user interface. The Workflow Agent itself only creates the main workflow and subflow agents and terminates everything at the end.

For WOS, there is also the challenge of having the workflow govern systems that communicate with non-agent means, such as distributed event-based systems and Web services. Several critical use cases will be defined for testing the AWS, OPS and WOS implementations separately and for testing them together. Stubs for other services will be provided as needed.

4.2 Overview of the TITAN Services

Implementation done for this thesis built on previous work and primarily focuses on developing the WOS Service and integrating it with the AWS and OPS Services, which were implemented in a previous phase of the project.

The JADE agent framework and WADE are used to implement the WOS service of TITAN. One JADE agent, the Monitor Agent, and two WADE agents, the Orchestration Agent and the Workflow Agent, provide the WOS service. Two JADE agents, the Message Agent and the Update Agent, provide the OPS service, and the Dispatcher Agent provides functionality for the AWS service with some help from the Event Cluster Assessor Agent. Along with these agents, there are also JMS Gateway agents and WSIG agents associated with each service to support communication. Figure 12 represents the agents in the three services (AWS, OPS and WOS Service).

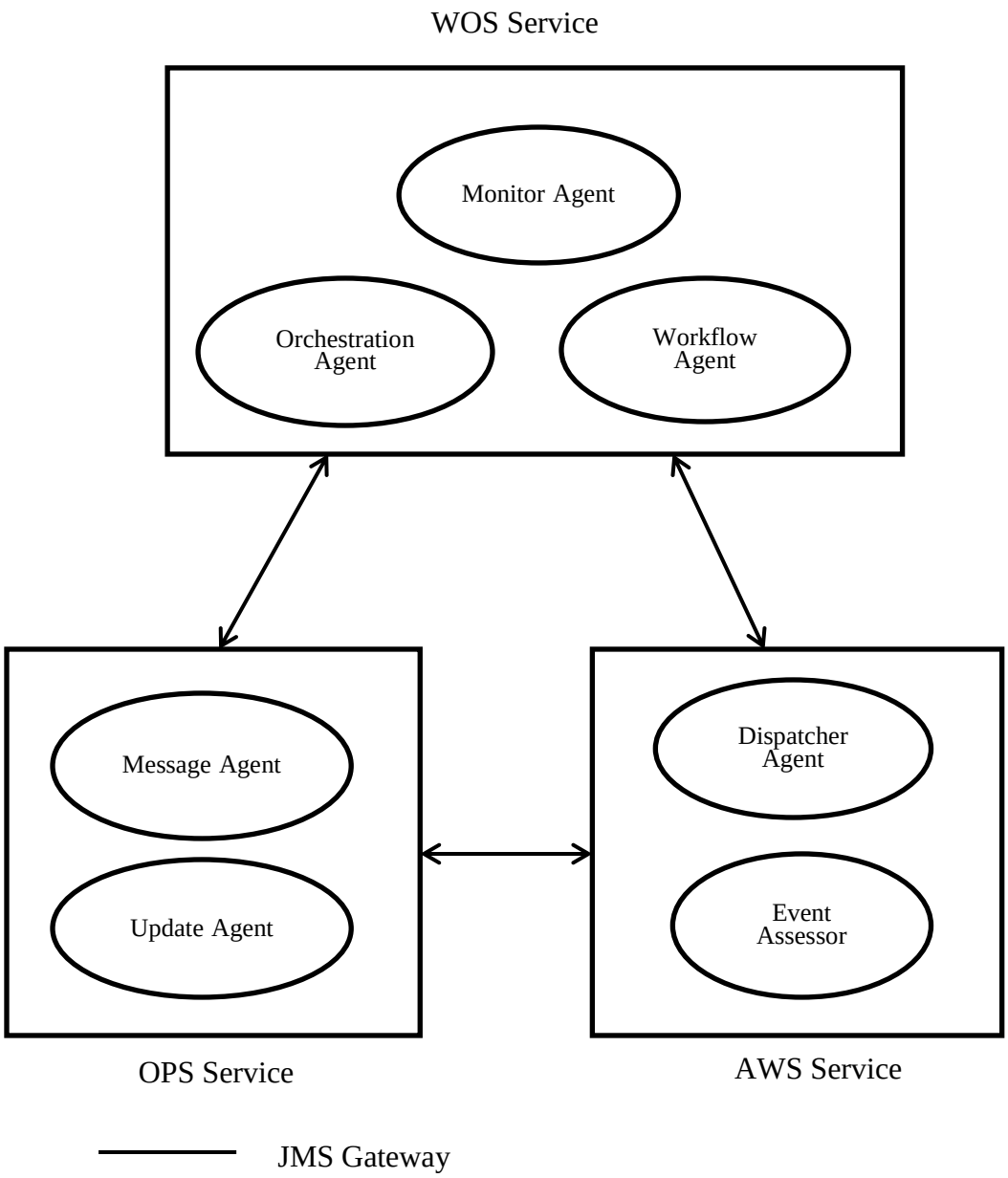

Figure 12. Communication between TITAN Services.

JADE agents use JMS-Agent Gateway APIs in order to create, send, receive and read JMS messages, allowing the agents to use the publish/subscribe messaging model for communication. The Monitor Agent, Message Agent, Update Agent, and Dispatcher Agent use this model for communication by publishing messages and subscribing to various JMS topics and the publish/subscribe model provides an event-based approach for communication and loose

coupling among the communicating entities.

In the WOS service, the Monitor Agent receives all incoming messages from the two Gateway Agents and directs all outgoing messages to the correct Gateway Agent. It maintains an archive of message versions to support status requests and situational reasoning.

In the OPS service, the Message Agent is responsible for handling the JMS messaging. It subscribes to three JMS Topics, `TITANcontrol`, `TITANassess`, and `TITANblackboard`, and is a publisher to two topics, `TITANexecute` and `TITANblackboard`. The Update Agent (also in the OPS service) subscribes and publishes to `TITANblackboard`. Initially the Message Agent receives the OPLAN/OPORD from the `TITANcontrol` topic and publishes the same on the `TITANexecute` topic. Subsequently, it receives all the updated OPLANs and OPORDs from the `TITANblackboard` topic, which are then published to `TITANexecute` so that the other agents are aware of all the updates made to the order. In addition, the Message Agent receives alerts and warnings from the `TITANassess` topic and forwards them to the Update Agent using the `TITANblackboard` topic. The Update Agent then sends a SOAP request to consume the Web service provided by the AWS service. The Update Agent receives an alert or warning report in the form of a SOAP response message, and, depending on the severity of the alert, it makes the necessary updates to the OPLAN/OPORD. A graphical user interface has been developed to display the current version of the OPLAN/OPORD in a tree view display.

Recall that the current implementation of the AWS service consists of a Dispatcher Agent and an assessor agent called the Event Cluster Assessor Agent. The Dispatcher Agent and the Assessor Agent together are responsible for generation of necessary alert and warning reports. The Dispatcher Agent receives the OPLAN/OPORD by subscribing to the `TITANexecute`

topic. In order to perform its monitoring and coordinating functions, the execution of the OPLAN/OPORD is simulated with the help of a user interface. The user interface takes input to simulate the occurrence of an alert situation available to the Dispatcher and the Assessor Agents. The Event Cluster Assessor Agent generates an Alert Report of the appropriate severity depending on the number of clusters of Improvised Explosive Devices (IEDs). The number of IEDs found is given as input by the user. The generated alert report is communicated to the Dispatcher Agent using ACL communication. The Dispatcher Agent notifies the Message Agent of the OPS about the alert by publishing a message to `TITANassess` topic. The WSIG agent allows the agent service provided by the Dispatcher Agent to be exposed as a web service so that it may be consumed by the OPS service agents.

4.3 Publishing and Subscribing Message using the JMS Agent Gateway

The JMS Agent Gateway allows JADE agents to communicate with other JADE agents located in various TBSs using JMS topics. To activate the Gateway, an instance of the JMS facilitator agent must be created. The JMS facilitator agent registers itself with the DF as type `JmsProxyAgent`. The facilitator agent is responsible for connecting and interacting with the JMS provider on behalf of its client agents. Using the facilitator agent, client agents can subscribe, publish messages to a JMS topic, and receive messages based on their subscription requests. The following section describes the ontology used by the JMS Agent Gateway to facilitate the JADE agents' use of JMS in an agent oriented fashion and, further, to assist them in publishing a message onto a JMS topic.

The ontology defines the vocabulary and the semantics used while sending and receiving a JMS message (Edward Curry, March 29, 2004). The ontology used here changes the task-oriented JMS API into goal-oriented interactions in the form of ACL messages. The ontology is

implemented by extending the class `Ontology` predefined in JADE. It contains a set of element schemas that describe the structure of concepts, actions, and predicates that are allowed as content of an ACL message. All agent interactions with the facilitator agent use the ontology as shown in Figure 13 shown below.

- Concept
  - AID
  - AgentAction
    - Unsubscribe
    - Subscribe
    - PublishMessage
    - OnMessage
  - JmsMessage
  - Subscription
  - ProviderInfo
  - Property

Figure 13. JMS Agent Gateway Ontology (Edward Curry, March 29, 2004).

The JADE agents using the JMS gateway ontology implement the `PublishMessage` task of the ontology to publish a message onto a topic. The `PublishMessage` task of the ontology directs the JMS facilitator agent to publish the messages contained within the `JmsMessage` frame to the JMS destination defined in the `ProviderInfo` frame. These frames contain all the required information needed to publish the message, allowing the agent to direct the facilitator's actions when processing the message.

A client agent uses the previously mentioned ontology to subscribe and receive subscription messages from a JMS Topic. The steps involved are: Create a `Subscribe AgentAction` and set the `ProviderInfo` of the action, specifying the destination type (`Util.QUEUE` or `Util.TOPIC`), the provider's URL, and the provider's initial context factory. Set the `Subscription`, specifying the destination and the subscription's durability and add a new `JmsSubscriptionInitiator`, passing in the above subscription message

4.4 Exposing Agent Service as a Web Service

Recall that WSIG (Weiss, 1999) offers interconnectivity that connects Web services to JADE platforms. WSIG exposes services provided by JADE agents and published in the JADE DF as Web services with no or minimal additional effort. This involves the generation of a suitable WSDL file for each service-description registered with the DF and possibly the publication of the exposed services in a UDDI registry(UDDI Spec Technical Committee).

All agents register with the DF so that they can be located by other agents. The agents register their agent services by providing a structure called the DF-Agent-Description that is defined by the FIPA specification. A DF-Agent-Description includes one or more Service-Description where each description describes a service provided by the registering agent. A Service-Description specifies one or more ontologies that must be known in order to access the published service. All actions that can be performed by the registering agent are defined in the specified ontologies. In order to expose an agent service as a web service, the Service-Description's `wsig` property must be set to `true` during the DF registration time.

4.5 Implementation of the WOS Service

Recall that the WOS service of TITAN consists of one JADE agent, called the Monitor Agent, along with two WADE agents, the Orchestration Agent and the Workflow Agent. It also has two gateway agents, namely, the JMS Gateway Agent and SOAP Gateway Agent used for communication.

The Monitor Agent generates a `PUBLISH` task from a `MONITOR-FOR-RESPONSE` task. The Monitor Agent also forwards `MONITOR` and `RESPOND-TO` tasks received from the Gateway Agents to the Orchestration Agent after processing, and forwards `PUBLISH` tasks to the target Gateway Agent. The Orchestration Agent receives tasks and other trigger objects and it

generates an `EXECUTE` task and assigns it to the Workflow Agent. A workflow from the repertoire of workflows is instantiated by the Workflow Agent to realize the assigned `EXECUTE` task.

The JMS Gateway Agent subscribes to the TITAN JMS topics and publishes to those topics. It provides a bi-directional communication interface between the WOS service and other TITAN TBS services. The SOAP Gateway Agent provides a SOAP service accessible by clients external to the TITAN TBS community. This agent can register its SOAP services with the jUDDI registry. When a SOAP call is invoked by an external client, this agent generates tasks to trigger the WOS service to fulfill the SOAP request.

Figure 14 represents the planned implementation of the WOS service prototype. Much of the detail becomes relevant now, in discussing how the Orchestration Agent fits into the picture. OPORDs are valid with respect to CERDEC's *prdC2* schema, but OPORDs actually used for testing are kept simple, and information relates mainly to the child units and the sequence of activities they perform. The Orchestration Agent uses JAXB to access the information in the OPORD.

In the planned system, for each subunit, the Orchestration Agent produces a Java code file defining the class for the proxy agent for that subunit. The behavior for the class will be a workflow in which the activities for each phase will be executed in sequence. The Orchestration Agent also produces a Java code file for the class for the main Workflow Agent. The behavior for this class is a workflow that has the behaviors of the proxy agents as subflows. After the Orchestration Agent produces these Java files, it sends a message to the Workflow Agent with the names of files produced. The Workflow Agent then compiles the files and produces agents as instances of the class.

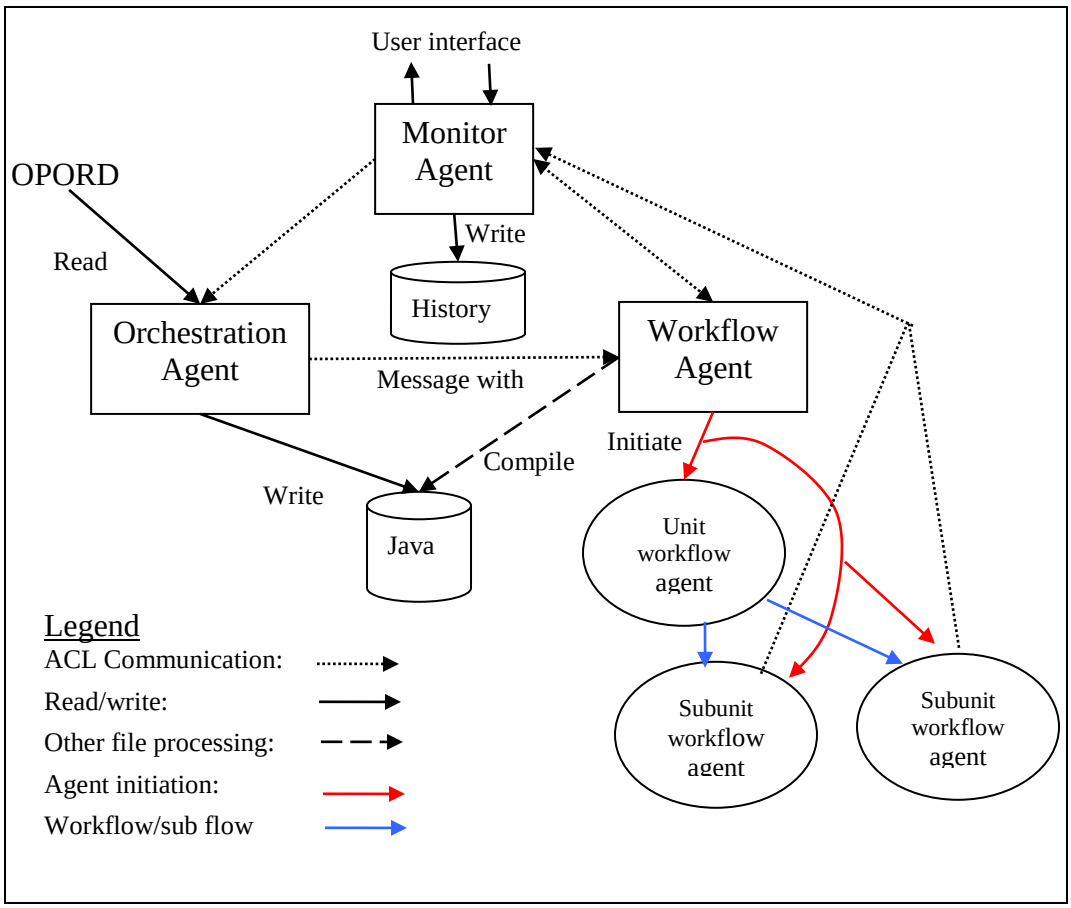

Figure 14. Design of the WOS Prototype.

The actual implementation has a Monitor Agent that acts as a communicator between agents. It sends and receives messages in the WOS service. Hash maps are used to maintain a history of actions and decisions. The Orchestration Agent and Workflow Agent can refer to this history in an attempt to find any previous situation similar to the one currently faced. The Orchestration Agent has an OPORD (an XML document) as an input. It uses JAXB to access the OPORD components, and it assigns the OPORD to the Workflow Agent. The Orchestration Agent as currently implemented searches for a workflow Agent for execution of workflow. If the Orchestration Agent finds any agent that is ready to perform workflow, it assigns the task to it; otherwise, it sleeps for a while and again searches for an agent. If an agent is found, then the Orchestration Agent assigns the task to the agent and traces the workflow. Here the agent that the

Orchestration Agent finds becomes the Workflow Agent. The main Workflow (unit) agent and the subflow (subordinate unit) agents communicate directly with the Monitor Agent. The Workflow Agent itself only creates the main workflow and subflow agents and terminates everything at the end.

TBSs of peers and children and the agents making them up will not be directly included in the workflow of a given TBS because a unit of significant size would require a prohibitively large and involved workflow. Also, the communication between TBSs is done with JMS. So a proxy agent is introduced for each peer and child TBS. Each proxy agent includes its own subflow. Such a subflow could involve further proxy agents for subunits two echelons down, but we stop one echelon down as direct communication stops one echelon down

4.6 Integrating WOS with AWS and OPS Services

A more comprehensive description of the planned implementation is as follows. The workflow in the WOS service is provided by the Workflow Agent. The Monitor Agent acts as an interface to simulate all communication into and out of the WOS. Much of this communication will be with the other TBSs, but it will also include reports from the AWS in the current TBS and updates to the OPORD from the OPS also in the current TBS. The original OPORD will be provided directly to the Orchestration Agent. Communication that would go to the parent will simply be displayed or stored on file. Updates that would come from the parent will be supplied as user input. This approach provides a clear partition of the system. A history of the unit's message activity will be maintained by the Monitor Agent. The Orchestration and Workflow Agents can query this history when a decision requires past values. The history can also be queried from outside the WOS. The history will be critical for the Orchestration and Workflow Agents when a work flow is aborted since they will be able to tell where the workflow is left off.

The actual implementation done here is as follows. The Monitor Agent keeps track of alerts and warnings provided by the AWS service using JMS messages. Both the Monitor Agent and the Message Agent are subscribed to the Dispatcher agent using `SubscribeMessage` topic The Dispatcher Agent in the AWS service sends messages using the `PublishMessage` topic to the Monitor Agent of the WOS service and the Message Agent of OPS service. In order to send messages to the Monitor Agent of WOS service, the AWS service was slightly modified from the form previously done by MS student Krishnamurthy (2010), where the AWS service communicated only with the OPS service The Orchestration Agent of the WOS service gets an OPORD from the user. The OPORD is read by the Orchestration agent, which assigns the task to the Workflow Agent, and the necessary changes in the OPORD are passed on to the Message Agent of the OPS service using the `TITANcontrol` topic. The Message Agent sends the OPORD to the Update Agent, and the Update Agent updates it as necessary using the `Update_Task` method. The `Update_Task` method unmarshalls XML document using JAXB and updates the facts and marshalls the formatted output to an XML document.

4.7 Configuration Issues

This section describes various issues that were encountered while configuring workstations for implementation of the TITAN services in the previous phase of the development done by previous MS student Krishnamurthy (2010) and in the current phase.

In the previous development, after installing JBoss version 5.1.0, the JMS-Agent Gateway add-on failed to recognize the JBoss JMS provider. To correct the issue, previous versions of JBoss (version 5.0.1.GA and 4.2.2.GA) were installed and found to work after several jar files (namely, `jbossall-client.jar` and `jbossjmx-ant.jar`) from the JBoss server distribution were copied and placed in the `lib` folder of JMS-Agent Gateway

distribution. It was also noted that the JMS-Agent Gateway add-on was pre-configured to work with the OpenJMS server rather than the required JBoss application server. To resolve this issue, the `JMS_Util.java` source file of the JMS-Agent Gateway distribution was modified to set the appropriate configurations that are required to connect to a JMS provider in the JBoss server; the whole distribution was rebuilt using Apache Ant.

The next issue in the previous development was with the WSIG agent. The WSIG agent failed to deploy because of conflicts in the versioning of some of the jar files in the WSIG distribution and ones in the JBoss distribution. To correct this, some of the conflicting jar files (namely, `xerces-2.6.2.jar` and `xml-apis.jar`) in the `lib` folder of the WSIG distribution were removed and the distribution was rebuilt using Apache Ant.

In the current development, the latest version of JBoss (7.0.0 Beta) was installed, but the JMS Agent Gateway failed to recognize the JBoss JMS provider. So an older version used by the previous MS student, version 4.2.2GA, was used. Keeping in mind the issue raised with WSIG in the previous development, it was correctly installed now, taking the necessary precautions.

There was an issue with the JMS Agent Gateway, which failed to work and some jar files (`beangenerator.jar` and `log4j-1.2.16.jar`) were copied from the JBoss server distribution to the `lib` folder of the JMS Agent Gateway, and class path was updated in the environment variables to correct the issue.

The other issue was with the JADE agent platform port number. A specified port number is given to correct the issue rather using the default port number.

CHAPTER 5

Discussion

The principle concern here is with the condition for coordination among TBSs in the TITAN prototype and, more extensively, with the condition for coordination in the full TITAN system, which would encompass the current prototype and considerably more. The critical notion is that of common knowledge, which can be shown to be a prerequisite for coordination. This chapter begins with a section that presents epistemic logic (Hendricks & Symons, 2009), the logic of knowledge, in which a reasonably adequate notion of common knowledge can be expressed. Section 2 outlines how the logic presented in section 1 can be applied to communicating concurrent systems. The third section distinguishes two types of common knowledge, common state knowledge (or CSK), which relates to the evolution of a given situation, and scaffolding, which relates to the social structures that enables coordination. The next section introduces notions of protocol and context to provide a formal means to express communication, action, and the evolution of a system. Section 5 sketches the protocols and context that establish and maintain CSK in TITAN, and the sixth section sketches protocols and context for the exchange of C2 products, which amounts to updating the scaffolding and thus requires guarantees for coordination. Section 7 discusses theory of mind (introduced in Section 2.6.2) from developmental psychology, indicates how the implementation of the WOS service is an analogue of theory of mind, and discusses how this helps maintain CSK in the TITAN prototype. This chapter is concluded by stating the advantages of various paradigms and their integration.

The term “agent” is used extensively in this chapter, not to denote agents in the narrow technical sense (such as JADE agents), but to denote anything that has state and can perform

actions. The discussion is generally appropriate for agents thought of as processes in a distributed computing system.

5.1 Epistemic Logic

This section is partitioned into two subsection. The first presents the syntax and semantics of epistemic logic, and the second address axiomatic systems for epistemic logic and the relation between what is provable in such systems and what turn out to be truths of logic according to the semantics. This section draws heavily on chapter 1,2 and 3 of Fagin et al. (1995)

5.1.1. Basics. Suppose we have *n* agents, named 1, …, *n*. Assume they all reason about a world described in terms of a nonempty set Φ of primitive propositions, *p*, *p′*, *q*, *q′*, ….. The semantics of the language of epistemic logic is quite straightforward. We augment the language with modal operators K_1, …, K_n (one for each agent). K_1 φ is read "agent 1 knows φ." A language is a set of (well-formed) formulas. To define a language for epistemic logic, we begin with the set Φ and declare inductively that,, if φ and ψ are formulas, so too are ¬ φ, (φ ∧ ψ), and K_i φ, for *i* = 1, ..*n*. Standard abbreviations are used for the usual connectives.

For the formal semantics for epistemic logic, we need a rigorous way to characterize the conditions under which a formula in the language is true. The guiding intuition here is that someone knows φ if and only if they can rule out any state of affairs ("possible world") in which φ is false.

A Kripke structure *M* for *n* agents over Φ is a tuple (S, π, K_1, …, K_n), where S is a set of states or possible worlds, π is an interpretation associating with each state in *S* a truth value for the primitive propositions of Φ (i.e., π(*s*): Φ → {true, false} for each $s \in S$), and K_i is a binary relation on *S*, i.e., a set of pairs of elements of *S* where $(s, t) \in K_i$ if agent i considers world *t* possible given his information in world *s*. We define the "truth relation" (*M*, *s*) |= φ, read "φ is

true at state s in Kripke structure M," inductively on the structure of proposition φ. The base case is given by π when φ is a primitive proposition p:

$(M, s) \models p$ (for $p \in \Phi$) iff $\pi(s)(p) = true$

The induction step handles the connectives $\wedge$ and ¬ and the operator K_i.

$(M, s) \models \varphi \wedge \psi$ iff $(M, s) \models \varphi$ and $(M, s) \models \psi$.

$(M, s) \models \neg \varphi$ iff $(M, s) \not\models \varphi$. (So the logic is two-valued)

$(M, s) \models K_i \varphi$ iff $(M, t) \models \varphi$ for all t such that $(s, t) \in K_i$.

The language is augmented with the modal operators E_G ("everyone in the group G knows") and C_G ("it is common knowledge among the agents in G"). Let $E_G{}^k$ be the E_G operator iterated k times We can now extend the definition of truth:

$(M, s) \models E_G \varphi$ iff $(M, s) \models K_i \varphi$ for all $i \in G$

$(M, s) \models C_G \varphi$ iff $(M, s) \models E_G{}^k \varphi$ for all $k = 1, 2, ..n$. (This is effectively an infinite conjunction)

A more intuitive, and ultimately useful, definition is based on the following concept. A state t is defined to be *G-reachable from state s in k steps* ($k \geq 1$) if there exist states $s_0, s_1, \ldots s_k$ such that $s_0 = s$, $s_k = t$, and for all j, $0 \leq j \leq k$-1, there exists $i \in G$ such that $(s_j, s_{j+1}) \in Ki$. Further, t is *G-reachable from s* if t is G-reachable from s in k steps for some $k \geq 1$ (i.e., if there is a path from s to t whose edge is labeled by members in G). We have the following results:

$(M, s) \models E_G{}^k \varphi$ iff $(M, t) \models \varphi$ for all t G-reachable from s in k steps.

$(M, s) \models C_G \varphi$ iff $(M, t) \models \varphi$ for all t G-reachable from s.

The specific content of a logical system is generally considered to be those formulas that are true independently of mundane facts—formulas we call *valid*—a well as rules of inference

that preserve validity. Epistemic logic provides a natural notion of validity, as true in all possible worlds. As we next show, this can be defined relative to a single Kripke structure or relative to a family of Kripke structure (for example, all those where the possibility relations are equivalence relations); validity can also be defined absolutely, without restriction to particular kinds of Kripke structures (hence cover all possible cases). Given Kripke structure $M = (S, \pi, K_1, \ldots, K_n)$, then, φ is *valid in M* (written $M \models \varphi$) if $(M, s) \models \varphi$ for all $s \in S$ and φ is *valid* (written $\models \varphi$) if φ is valid in all structures.

It is particularly appealing for possibility relations to be equivalence relations, and certain formulas valid for structures with such relations are central to the literature on epistmic logic. For formulas φ and ψ,·all structures M where each possibility relation K_i is an equivalence relation, and for agents agents $i = 1, \ldots, n$., the following are the formulas in question.

- $M \models (Ki\ \varphi \wedge Ki\ (\varphi \Rightarrow \psi)) \Rightarrow Ki\ \psi$ (Distribution Axiom)
- If $M \models \varphi$, then $M \models Ki\ \varphi$ (Knowledge Generalization Axiom)
- $M \models Ki\ \varphi \Rightarrow \varphi$ (Knowledge Axiom or Truth Axiom)
- $M \models Ki \Rightarrow Ki\ Ki\ \varphi$ (Positive Introspection Axiom)
- $M \models \neg\ Ki\ \varphi \Rightarrow Ki \neg\ Ki\ \varphi$ (Negative Introspection Axiom)

Concerning the E_G and C_G operators, for all formulas φ and ψ, all structures M (not just those whose possibility relations are equivalence relations), and all nonempty $G \subseteq \{1,.., n\}$,

- $M \models E_G\ \varphi \wedge_{i \in G} K_i\ \varphi$
- $M \models C_G\ \varphi \Leftrightarrow E_G\ (\varphi \wedge C_G\ \varphi)$ (The Fixed-Point Axiom)

The second statement here says that $C_G\ \varphi$ can be viewed as a fixed point of the function

$$f(x) = E_G\ (\varphi \wedge x),$$

which maps a formula x to the formula $E_G(\varphi \wedge x)$.

5.1.2. Soundness and completeness of axiomatizations of epistemic logic. Since at least the time of Euclid, a goal of mathematics has been to express theories as axiomatic systems, and this goal has been pursued for epistemic logics. An axiom *system AX* consists of a collection of formulas called *axioms* and *inference rules*, each of the form "from $\varphi_1, \ldots, \varphi_n$ infer ψ", where $\varphi_1, \ldots, \varphi_n$ and ψ are formulas. A proof is a *proof of the formula* φ if the last formula in the proof is φ. φ is *provable* in *AX*, written $AX \vdash \varphi$, if there's a proof of φ in *AX*.

The following are the axioms and inference rules of the epistemic logic $S5_n$ (where the subscript n indicates the size of the set of agents). Some of these correspond to what earlier were said to be formulas valid for Kripke structures whose possibility relations K_i are equivalence relations. Axioms (except for A1) are labeled with their traditional labels, but the rules are identified simply as R1 and R2. Traditional names are shown in parentheses.

A_1. All tautologies of propositional calculus.

K. $(K_i\varphi \wedge K_i(\varphi \Rightarrow \psi)) \Rightarrow K_i\psi$, $i = 1,\ldots,n$ (Distribution Axiom)

T. $K_i\varphi \Rightarrow \varphi$, $i = 1, \ldots, n$ (Knowledge Axiom)

4. $K_i\varphi \Rightarrow K_i K_i \varphi$, $i = 1, \ldots, n$ (Positive Introspection Axiom)

5. $\neg K_i\varphi \Rightarrow K_i \neg K_i \varphi$, $i = 1, \ldots, n$ (Negative Introspection Axiom)

R_1. From φ and $\varphi \Rightarrow \psi$ infer ψ. (Modus Ponens)

R_2. From φ infer $K_i \varphi$. (Knowledge Generalization)

An axiomatic system generally is developed to capture in a rigorous way a pre-existing theory or least intuitive understanding of some domain. On a general level, given a language L (generally by a recursive definition of well-formed formula), one may seek an axiomatic system in which are provable those formulas in L one would generally accept in the theory while those formulas in L not thus acceptable are not provable. But we have already seen a rigorous way to

distinguish formulas in the language, namely, by way of validity. An axiom system *AX* is *sound* for a language ***L*** with respect to a class ***M*** of structures if every formula in ***L*** provable in *AX* is valid with respect to ***M***. The system *AX* is *complete* for ***L*** with respect to ***M*** if every formula in ***L*** that is valid with respect to ***M*** is provable in *AX*. *AX* characterizes the class ***M*** if it provides a sound and complete axiomatization of that class.

It can be shown that $S5_n$ is a sound and complete axiomatization of the class of Kripke structures whose possibility relations are equivalence relations Various different sets of axioms (along with the rules R_1 and R_2) have been shown to provide sound and complete axiomatizations for classes of Kripke structures whose possibility relations enjoy properties other than the three properties characterizing equivalence relations. $S5_n$ is the most commonly used epistemic logic and, in fact, is stronger (has more provable theses) than the usual alternatives.

If to $S5_n$ are added the following axioms and rule of inference, we get an axiomatic system $S5_n^C$ that is a sound and complete axiomatization for our epistemic-logic language enhanced with the E_G and C_G operators with respect to the family of Kripke structures whose possibility relations are equivalence relations. (In fact, the following are simply valid, that is, with respect to all Kripke structures.)

- C_1. $E_G\ \varphi \Leftrightarrow \wedge_{i \in G} K_i\ \varphi$
- C_2. $C_G\ \varphi \Rightarrow E_G(\varphi \wedge C_G\ \varphi)$
- R_{C1}. From $\varphi \Rightarrow E_G\ (\psi \wedge \varphi)$ infer $\varphi \Rightarrow C_G\ \psi$ (Induction Rule)

5.2 Knowledge in systems

We consider a collection of interacting agents a system. Systems thus include, for example, societies (where the agents are people) and distributed systems (where the agents are processes in a computer network running a given protocol). A military hierarchy is also a system

(with a particular social structure). In the context of TITAN, we focus on the TBSs in a military hierarchy as a convenient abstraction. In fact, we abstract considerably so that the concepts developed in (Fagin et al., 1995) may be applied to gain analytic leverage. The initial challenge is to characterize systems so that epistemic logic as developed in the previous section may be applied. This section draws on chapter 4 of Fagin et al. (1995).

In a system, then, we assume that agent i has a set L_i of *local states*. An agent's local state encapsulates all the information to which it has access. Besides agents, a system has an *environment*, which has its own set L_e of local states. A *global state* of a system with n agents is an (n+1)-tuple (s_e, s_1, …, s_n), where s_e is the state of the environment and s_i is the state of agent i. The set $\mathcal{G}$ of global states, then, is $L_e \times L_1 \times \ldots L_n$. A *run r* over $\mathcal{G}$ is a function from time to global states. Time ranges over the natural numbers, so a run is a sequence of global states. Time is measured by a clock external to the system, and an agent need not know that the time is, say, m. If it does, this must be encoded in its local state. For run r and time m, (r, m) is a *point*. Where $r(m) = (s_e, s_1, \ldots, s_n)$ is the global state at point (r, m), we define $r_e(m) = s_e$ and $r_i(m) = s_i$ for i = 1, …, n . *Round m* in run r takes place between times m-1 and m (so round 1 is the first round). An agent is viewed as doing an action during a round. A *system* $\mathcal{R}$ *over* $\mathcal{G}$ is a nonempty set of runs over $\mathcal{G}$.

A statement such as "R does not know φ" means that, as far as R is concerned, the system could be at a point where φ does not hold. R's knowledge is determined by its local state. The critical point that is exploited here to apply epistemic logic is that R cannot distinguish between two points of the system where it has the same local state but can distinguish points where its local states differ. A system can be viewed as a Kripke structure except that there is no function π assigning truth values to propositions.

A set Φ of primitive propositions is assumed for describing basic facts about the system and one introduces the notion of an *interpreted system I*, which is a pair $(\mathcal{R},\pi)$, where $\mathcal{R}$ is a system over a set $\mathcal{G}$ of global states, and π is an interpretation for the propositions in Φ over $\mathcal{G}$, which assigns truth values to the primitive propositions at the global states. So, for every $p \in \Phi$ and $s \in \mathcal{G}$, we have $\pi(s)(p) \in \{true, false\}$. π also induces an interpretation over the points of $\mathcal{R}$ if we let $\pi(r,m)$ be $\pi(r(m))$.

An interpreted system $I = (\mathcal{R},\pi)$ is associated with Kripke structure $M_I = (S,\pi,K_1,\ldots,K_n)$, where S consists of the points in I and $K_1,\ldots,K_n$ are binary relations on S, with K_i taken to be the relation $\sim_i$ defined as follows. If $s = (s_e,s_1,\ldots,s_n)$ and $s' = (s_e',s_1',\ldots,s_n')$ are two global states in $\mathcal{R}$, then, if i has the same state in both s and s' (i.e., $s_i = s_i'$), then s and s' are *indistinguishable to agent i*, written $s \sim_i s'$. (There is no possibility relation K_e for the environment as there is no interest in what the environment knows). The indistinguishability relation $\sim_i$ is extended to points by taking two points (r,m) and (r',m') to be *indistuinguishable* to i, written $(r,m) \sim_i (r',m')$, if $r(m) \sim_i r'(m')$. Clearly, $\sim_i$ is an equivalence relation.

A formula can now be defined to be true at a point (r,m) in an interpreted system I by applying the earlier definitions to the related Kripke structure M_I. In general, $(I, r, m) \models \varphi$ iff $(M_I, s) \models \varphi$, where $s = (r, m)$. Specifically,

$(I, r, m) \models p$ iff $\pi(r, m)(p) = true$ for $p \in \Phi$.

$(I, r, m) \models K_i \varphi$ iff $(I, r', m') \models \varphi$ for all (r', m') such that $(r, m) \sim_i (r', m')$.

For formulas involving common knowledge, the previously defined notion of G-reachability is used. Note that the truth of formulas of the form $K_i \varphi$ depends only on the local state of i while the truth of those of the form $C_G \varphi$ depend on the global state. It is just the global state at the

point (r, m) unless φ includes temporal operators, in which case the truth could depend on global states potentially anywhere on the run r.

5.3 Common Knowledge in TITAN

In a hierarchy of TBSs, an OPORD amounts to a coordination script. The OPS service receives an OPORD from the parent TBS and fills with the detail needed to direct and coordinate its child units. OPORDs are provided to the TBSs for subordinate units and the TBS as a whole directs the subunits in carrying out these parts of the mission. Detail is filled out as the script is disseminated down the hierarchy. Each TBS is free to determine how its unit will carry out its part of the mission within the constraints of the current OPORD. How its unit carries out its part depends critically on the coordination scripted into the OPORDs for its subunits and monitored by the TBS itself.

The concept that rises to salience here is that of common knowledge, a necessary condition for coordinated action. Several paradoxical conclusions follow in epistemic logic (Fagin et al., 1995): we know all logical consequences of what we know (logical omniscience), and information becomes shared in the required sense at the same time for all agents sharing it. These are paradoxes if common knowledge is regarded as a disposition of individuals, but, as Barwise points out, common knowledge is not properly knowledge: knowing that φ is stronger than carrying the information that φ since it relates to the ability to act. He concludes that common knowledge (as per fixed the-point approach) is a necessary but not sufficient condition for action and is useful only when arising in a shared situation that, if maintained, "provides a stage for maintaining common knowledge."

Barwise addresses situations, but common "knowledge" is more embracing. H. H. Clark and Carlson (1982) identified three "co-presence heuristics" giving rise to different kinds of

shared “situations”. Two, physical co-presence (e.g., shared attention) and linguistic co-presence (as in conversation), properly relate to situations, but the third, community membership, is not temporally or spatially restricted and is presupposed by the other two.

Common knowledge is characterized due to community membership as the social part of what A. Clark (1998) called scaffolding: a complex world of physical and social structures on which the coherence and analytic power of human activity depends. In contrast, let “common state knowledge”, abbreviated as “CSK”, refer what is established by Clark and Carlson’s two non-scaffolding co-presence heuristics, which grows as members of the group communicate and share their environment. Common knowledge, rather than being a disposition mysteriously shared by a group, is either scaffolding or a self-referential feature of the situation.

Some influential work in psycholinguistics (H.H Clark, 1996) emphasizes CSK (or mutual belief) in conversation. In focusing on the detailed interaction (seen as negotiation over content) in conversation, the notion of common ground was introduced by Clark as “shared information or common ground—that is, mutual knowledge, mutual beliefs, and mutual assumptions” (Hebert. H Clark & Brennan, 1991).

Common ground has found application in shared situation awareness, but the simple self-referential aspect of common knowledge is generally lost in the detail, and common ground is characterized as a shared perspective ("Introduction," 1996) or (simply) shared knowledge, beliefs, and assumptions (Nofi, 2002), or the negotiation is explicated in terms of state transitions (Pérez-Quiñones & Sibert, 1996) with no focus on self-reference. In the context of TITAN, the protocols and ontologies are designed into the TBSs as scaffolding, and the XML documents that are communicated among the TBSs are essentially coordination scripts. TITAN CSK is similar to common ground, but communication among TBSs in TITAN is scripted and has larger scope

in the sense that message syntax and context are understood as intended because of the correctness of the underlying protocol and schema.

The notion of common knowledge has proved useful in a wide array of disciplines. It was introduced in the 1960s by the philosopher David Lewis and has been used in the analysis of protocols (scaffolding) in distributed systems (Fagin et al., 1995) with an eye to CSK as the distributed state evolves. CSK also arises from more scripted activities, such as the rituals analyzed by Chwe (2001), who demonstrated that rituals are rational at a meta-level since they establish common knowledge enabling coordinated actions with significant payoffs. In game theory, games of complete information assume that players' strategies and payoffs are scaffolding, while games of perfect information assume that players' moves are CSK. TITAN achieves CSK in much more flexible ways that coordinate complex activities over extended periods.

The self-referential nature of CSK is revealed by abstracting away the communication details. In designing a system (scaffolding) to maintain CSK, we should identify what must be included in the CSK and how the situations are self-referential. Almost always, however, these issues remain in the tacit understanding of the problem. And sometimes an analysis in terms of CSK is hardly called for, as in designing protocols for distributed systems, where experience with the context generally makes what is required explicit. But when the participants are autonomous, content relates to the real world, and the tasks are novel for the analyst, it is always a good idea to consider these issues.

5.4 Protocols and Contexts

Actions are nothing but sending and receiving messages, and possibly some internal actions. Following chapter 5 of Fagin et al. (1995), a *joint action* is taken to be a tuple of form

(a_e, a_1,.., a_n), where a_e is defined as action performed by environment and a_i is action performed by agent *i*. To make joint actions cause the system to change states, a *global state transformer* τ is associated with it, where τ is simply a function mapping global states to global states, i.e., τ: $\mathcal{G} \rightarrow \mathcal{G}$. Agents usually perform actions according to the *protocols* (rule for selecting actions) A *protocol* P_i for agent *i* is a function from the set L_i of agent *i*'s local states to nonempty sets of actions in ACT_i. It is deterministic if the sets are always singleton sets. A protocol for the environment is a function from L_e to nonempty subsets of ACT_e. A *joint protocol P* is a tuple $(P_1,\ldots,P_n)$ consisting of protocols P_i for each agent $i = 1, \ldots, n$. A *context* γ is defined as a tuple (P_e, $\mathcal{G}_0$, τ, Ψ), where

- P_e: $L_e \rightarrow 2^{ACT_e}$-{∅} is a protocol for the environment,
- $\mathcal{G}_0$ is a nonempty subset of $\mathcal{G}$,
- τ is a transition function, and
- Ψ is a condition on runs. Often Ψ consists of temporal logic formulas; the runs in Ψ are those satisfying these formulas

5.5 Protocols for CSK in TITAN

In modeling the behavior of a military hierarchy, the salient features relate to the coordinated behavior of the units as they follow their operational orders. This coordination requires CSK. We abstract so as to consider only the behavior of the TBSs as they "execute" their OPORDs, and we consider hierarchies only two-levels deep. We simplify by assuming that the OPORD for the parent (or "main unit," henceforth abbreviated "MU") contains as well-formed parts each of the OPORDs of its subordinates even though the OPS service of the MU TBS will generally elaborate the OPORD for a subordinate. We initially consider the OPORDs fixed, with no modification during mission performance due to the arrival of WARNOs or

FRAGOs. Finally, the behavior of a TBS is similar to that of a state machine: on receiving an input message, it changes state (while its unit is performing the indicated activity) and possibly sends an output message. A sharper picture of what is involved emerges in the following discussion. The analysis here ignores communication within the same echelon since, in general, interest is with how the MU coordinates its subordinates; in any case, any inter-echelon communication is subject to control by the MU.

A natural way to represent a global state for a TITAN two-level hierarchy with n subordinates labeled 1, 2, …, n is as an (n+2)-tuple (s_e, s_m, s_1, …, s_n), where s_e is the state of the environment, s_m is the state of the MU, and s_i is the state of subordinate i, i = 1, 2, …, n., which is the state it is currently in as per its OPORD. The state of the environment, s_e, relates only to the communication that transpires between the MU and the subordinates. For analysis, it is convenient to define it so that the context is a so-called "recording context," where the environment's state maintains a history of the joint actions. The state of the MU, s_m, is taken to be a pair (C, s_{mu}), where C is a counter (assuming values 0 to n) and s_{mu} is an n-tuple (s_{m1}, s_{m2}, …, s_{mn}), where s_{mi} is the current state of the MU's proxy for subordinate i. (s_{MU} is thus pictured as what is known as a *product automaton*.) The most obvious feature of successful coordination is that $s_{mi} = s_i$ for all i = 1, 2, …, n.

To expand the picture of how a run evolves, we consider each coordination point in an OPORD of a subordinate to correspond to two states. On completing an activity, the subordinate enters a `Done` state. Exiting this state, it sends the MU a `done` message and enters a `BWaiting` state. On receiving a `begin` message from the MU, the subordinate proceeds with the next activity. This sequence of states and transitions is depicted in Figure 15(a). In a corresponding way, the component of the product machine of the MU that relates to a given subordinate has

two states for each coordination point, as shown in Figure 15(b). After accounting for activity on the part of the subordinate, it enters a `DWaiting` state. When it receives a `done` message from the subordinate not only from the subordinate in question but from all *n* subordinates, it proceeds to a `Begin` state. On exiting this state, it sends a `begin` message to the subordinate. The fact that the MU does not make the transition from the `DWaiting` to the `Begin` state until it has received `done` messages from all subordinates testifies to the fact that the control goes beyond simply updating the product automaton as dictated by transitions in the individual components. That fact also accounts for the need for a counter. This transition abstracts over not only the tracking of `done` messages but also any processing the MU might do regarding timing.

In terms of protocols and contexts, the set ACT_i of actions for subordinate *i* consists of sending and receiving the various messages just noted. The actual carrying out of the unit's activities relates to what from the current point of view are internal actions. The set ACT_{mu} of actions of the MU are *n*-tuples of the form $(a_{m1}, a_{m2}, \ldots, a_{mn})$, where a_{mi}, $i = 1, \ldots, n$, is either sending a message to subordinate *i*, receiving a message from *i*, or Λ (do-nothing).

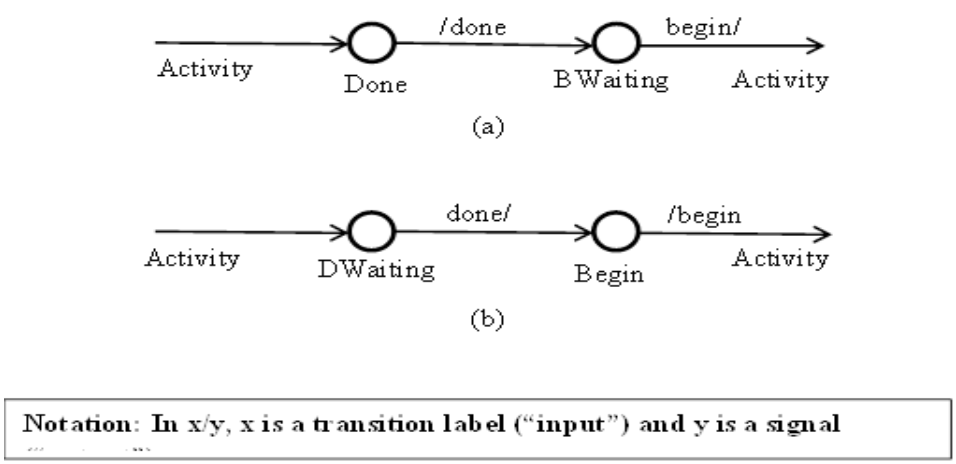

Figure 15. (a) A sequence for a subordinate for a coordination point, (b) The corresponding sequence for the MU.

We thus allow only one communication action between the MU and a given subordinate in a round, and so it takes a minimum of two rounds to exchange messages. We assume that a message (whether from the MU to a subordinate or from a subordinate to the MU) is reliably received in the round in which it is sent. This simplifying assumption is useful in drawing the focus away uninteresting details. We also assume that the MU sends begin messages to all subordinates in the same round. Our assumptions ensure that, when the subordinates begin a new phase, this is CSK among all TBSs. In particular, common knowledge cannot be attained without reliable communication (see (Fagin et al., 1995) Sec. 6.1), and these assumptions ensure the simultaneous change in the TBSs' knowledge that is required for achieving new common knowledge (see (Fagin et al., 1995), Sec. 11.2). The assumptions made to guarantee common knowledge focus interest on the OPORD-directed communication and the role of the proxies in the WOS service of a MU.

Continuing with the description of the context, a joint action here has the form (a_m, a_1, …, a_n), where $a_m \in ACT_{mu}$ and ai $\in ACT_i$ for $i = 1, \ldots, n$. The transition function τ is defined so that receipt of messages causes state transitions and increment of the MU's counter as outlined above. Similarly, the protocol for subordinate i is defined so that messages are sent on state transitions as outlined above. The set of initial global states, $\mathcal{G}_0$, here consists of those global states where all units are at their start states and the MU's product automaton records the corresponding situation. Finally, for Ψ, the condition on runs, we specify that every `done` message from a subordinate to the MU is followed within a fixed maximum number of rounds by a `begin` message from the MU to that subordinate. This implies a bound on the differences in the completion times of the subordinates' activities and thus also assumes that a mission in some sense succeeds as all subunits are assumed to reach a state in some way anticipated in the

OPORD in bounded time. These assumptions simplify the analysis and are warranted when missions are intended to complete within a fixed period.

5.6 Protocols for Sharing Scaffolding in TITAN

In addition to the TITAN behavior modeled as a system in which TBSs are coordinated by "executing" OPORDs and attaining CSK, there is another level of behavior in TITAN in which TBSs exchange C2 products (as XML documents). The most significant C2 products are OPORDS, specifying the operational orders for the unit and its subunits. Other C2 products include WARNOs, used to communicate a "heads-up", and FRAGOs, fragmentary orders for updating OPORDs. The structures of WARNOs and FRAGOs are compatible with the structure of OPORDs.

The operational orders in effect essentially define the rules of the game for the mission at hand and thus provide part of the scaffolding for the hierarchy, in particular, the more specific aspects of the scaffolding, which assume more general conventions and rules, including the protocol by which these C2 products themselves are communicated in ways in which they may figure in common knowledge.

A high-level review is in order here of how C2 products are sent around a TITAN hierarchy. We restrict ourselves to two-echelon TBS hierarchies. There are three basic scenarios, as follows.

- Initial Scenario: Initially, the OPS service of the MU receives an OPORD, and its Update Agent elaborates the operational orders for the subordinates, sending OPORDs for them, which are handled by the Update Agents of their OPS services. At both levels, the OPORD received is sent to the TBS's WOS service, where the Orchestration Agent constructs a workflow executed by the Workflow Agent.

- Warning Scenario: When the AWS service detects a concern, its Dispatcher Agent sends an alert or warning to the MU's OPS service, whose Update Agent constructs WARNOs for the subordinates. These are sent to subordinates' OPS services, where the Update Agents update their OPORDs accordingly. An updated OPORD is sent if to the WOS, whose Orchestration Agent updates the workflow executed by the Workflow Agent.
- Update Scenario: The Update Agent of the OPS service of the MU constructs FRAGOs for various reasons to modify the operational orders of one or more of the subordinates. These are sent to the OPS services of each subordinate TBS involved, where they are handled by the Update Agent, which modifies the current OPORD before sending it on to the TBS's WOS. As before, the Orchestration Agent updates accordingly the workflow executed by the Workflow Agent.

We consider receipt of a C2 product as implicitly involving an agreement between the sender and the receiver that they will abide by the operational orders therein contained or implied. This agreement is common knowledge among the two parties, for, as the following argument from ((Fagin et al., 1995) Chap. 6) shows, agreement implies common knowledge. So let $agree(\psi)$ be a formula true in states where the members of group G agree on ψ. If they have agreed on ψ, then surely all know that they have agreed on ψ. We thus expect $agree(\psi) \Rightarrow E_G(agree(\psi))$ to be valid for any ψ and any group G. But then, by the Induction Rule for common knowledge, it follows that $agree(\psi) \Rightarrow C_G(agree(\psi))$. That is, agreement implies common knowledge. We must, however, guarantee that these agreements are coherent, for example, that there is not an agreement between the MU one subordinate that conflicts with an agreement between the MU and another subordinate, and this is a coordination problem.

We conceptualize the commitment to a set of operational orders as a commitment to the activity thus mandated. The MU, then, has a representation of the commitment of each subordinate with which it has an active agreement. This is what has been called a meta-representation (Goldman, 2006), a representation of something that itself has a representational aspect. In fact, it is convenient to view what is established when scaffolding is exchanged as a meta-representation even for a TBS that has the commitment in question. That is, the TBS has a view of its own commitment (roughly, by the fact that its OPS service has handled it). The TBS with the commitment in question, of course, is also "executing" the OPORD, which is in the realm of CSK as previously discussed.

To characterize the exchange of C2 products in terms of protocols and a context, we consider a subordinate's state at this level to be its currently active OPORD along with an indication of whether it recognizes an active warning. The state of the MU at this level is likewise its currently active OPORD along with, for each subordinate, an indication of whether the agreement has been made with it and, if so, whether it should be under a warning. Coordination requires that either all agreements are in effect or none are. It also requires that subordinates recognize warnings they should be under and enter and exit warning states in a coordinated way (the simplest cases, assumed here, being where either all or none are in such a state).

A global state, as before, is an (n+2)-tuple consisting of the state of the environment, the state of the MU, and the states of the n subordinates. As before, we can take the state of the environment to be as required for a recording context.

Some of the actions of the TBS have already been indicated. In a two-echelon hierarchy, a subordinate can receive an OPORD, WARNO, or FRAGO, and it can send an alert or warning;

it can also do noting (Λ). For the protocols suggested below, subordinates must also be able to send acknowledgments and receive commit messages. The MU, correspondingly, can send an OPORD, WARNO, or FRAGO to any subordinate, and it can receive an alert or warning from any subordinate. It can also receive an acknowledgment from and send a commit to any given subordinate. And, indeed, it may neither send something to nor receive something from a given subordinate (Λ). As before, we take an action of the MU to be a tuple of the form (a_{m1}, a_{m2}, …, a_{mn}), with the components a_{mi} now as just discussed. A joint action, then, is an (n+1)-tuple consisting of actions by the environment and the n subordinates.

We again assume that communication is reliable and that messages are received in the same round in which they are sent.

We can now sketch out protocols for the TBSs for the three scenarios described above. The protocols for the TBSs, given the discussion of actions above, are implicit in the following as is the transition function.

The Initial Scenario can use a simplified version of the two-phase commit protocol. When a subordinate has a workflow for its OPORD running, it sends an acknowledgment to the MU. When the MU has acknowledgments from all subordinates, it simultaneously sends a commit message to all and sets its commit flag for all. It is convenient to assume that there is a time bound on how long this takes from start to finish.

The Update Scenario is the same as the Initial Scenario. The only difference is that the Update Agents must modify OPORDs that has already been committed to.

In the Warning Scenario, a subordinate sends an alert, and then the MU makes and sends FRAGOs for all subordinates (including the one sending the alert). When each has received its FRAGO and has the Workflow Agent executing the updated OPORD, it sends an

acknowledgment. When the MU has received acknowledgments from all subordinates, it simultaneously sets its warning flag for each subordinate and sends a commit message to all. When a subordinate receives a commit message, it sets its warning flag. This same scheme works for the case where only a subset of the subordinates needs warning. A similar protocol would work in principle for clearing warnings, but detecting an all-clear condition in practice would itself likely require coordination since “no danger” is a universal conditions, unlike “warning,” which is an existential condition.

It is quite straightforward to show that the suggested protocol guarantees common knowledge that the TBSs are all committed to the overall OPORD (Initiation Script), that they are committed to updates made to the operational orders (the Update Scenario), and that they have been warned and are committed to responding in the agreed way. Some of this common knowledge relies on having a sufficiently granular notion of time (as what counts as simultaneous depends on how finely we view time)—cf.[(Fagin et al., 1995), Sec. 11.4].

Common knowledge might not be that useful if it takes too long to attain. In the present case, when an alert is raised, the protocol might initially forego sending WARNOs and propagate simple alerts and their corresponding acknowledgments. This would establish common knowledge of an alert state, after which the operational implications could be communicated with WARNOs.

Regarding the context in which the joint protocol runs, we have a sketch of the environment’s protocol P_e and of the transition function τ. The set $\mathcal{G}_0$ of initial global states is the set of those global states that are possible before the initial OPORD is sent to the MU. Finally, no condition Ψ on runs is strictly needed although time bounds could facilitate analysis.

5.7 Theory of Mind in TITAN

Coordination of child TBSs by a parent TBS using the same coordination paradigm as used within a TBS—a multiagent system—is infeasible because of the communication requirements. So the WOS maintains proxy agents to model the children of the TBS. More specifically, the workflow maintained by the WOS contains agents that themselves implement workflows that mirror the salient aspects of the workflows of the children. The mission is partitioned into stages, and the sub-flows synchronize on stage transitions by receiving JMS reports sent to the TBS from the child TBSs and passed along to the WOS. The proxies can also respond to alerts from other TBSs and to updates to the OPLAN. The stages are identified in the OPORD, which also contains a script for each child. In addition, the WOS must have access to enough information from the parent TBS's OPORD to simulate the siblings of the TBS. The proxy agents for the siblings are not part of the WOS's workflow but are coordinated with it. Like the proxies for the children, JMS reports from their corresponding TBSs are sent so they may maintain up-to-date models. Finally, the TBS must be sent information on its parent's script so that the WOS may form a proxy for the parent, which again is not part of the workflow.

Use of proxies by the WOS implements a scaled-back version of what is known as theory of mind (ToM) in developmental psychology (Flavell, 2004). Theory of mind is the ability to attribute mental states (beliefs, desires, intentions, assumed roles, etc.) to oneself and others and to understand that others have mental states different from one's own. The WOS's version is scaled back since it is restricted to intentions and (to a lesser extent) beliefs and conflicting intentions do not arise. Early work in ToM focused on the striking improvement in children between ages three and five on false-belief (FB) and Level 2 visual perspective-taking (PT) tasks and similar tasks. As an example of an FB task, the child watches as a puppet sees a cookie put

in one of two boxes and leaves. In the puppet's absence, someone moves the cookie to the other box. When the puppet comes back, the older child (having a notion of false belief) says the puppet will look in the original box. The younger child (not having this notion) will say the puppet will look in the box with the cookie. As an example of a TB task, the older child, but not the younger, understands that a picture book oriented correctly for them on the table looks upside down to a person seated opposite. ToM development provides a scaffold for early language development: when a child hears an adult speak a word, they recognize that the word refers to what the adult is looking at. And it is suggested (Astington & Jenkins, 1995) that the connection of pretend play (involving role assignments) with false belief understanding is in the representation of the other's beliefs and goals, which often conflict with one's own. ToM is apparently at odds with CSK in emphasizing differences rather than what everyone knows, but note that CSK is involved in pretend play and denoting phrases, cases that involve cooperation. Tomasello (2009) uses the term "shared intentionality" for the underlying psychological processes that make possible our species-unique forms of cooperation. Shared intention lets us, in cooperative endeavors, "create with others joint intention and joint commitment," which "are structured by processes of joint attention and mutual knowledge."

The two accounts of theory of mind the predominate in the philosophical literature are simulation theory, according to which we understand the mental states of others by simulating them in ourselves, and theory theory, according which we understand these states by reasoning based on folk psychology (Goldman, 2006)

Wiriyacoonkasem and Esterline (2001) have shown how the physical co-presence heuristic may be used so that groups of artificial agents may attain common knowledge by perceptual means. To focus on the episodic nature of physical co-presence evidence, we introduced into

epistemic logic a modal operator Sat φ for agent a seeing at time t that φ. Time parameters introduced for epistemic operators, have the axioms $S_a^t \varphi \Rightarrow K_a^t \varphi$ and (for simplicity) $K_a^t \varphi \Rightarrow K_a^u \varphi$ for all u > t, and focus on cases where, for example, both $S_a^t S_b^t \varphi$ and $S_b^t S_a^t \varphi$ hold. The formalism exposes strong reasons to hold that, to attain CSK, agents must model each other's perceptual abilities, which requires common knowledge of shared abilities. A procedure is given for determining from the models for statements in this logic when CSK is attainable and developed a prototype system implementing the procedure enhanced with a neural network to capture hidden knowledge from a trainer. The linguistic co-presence heuristic can in principle be handled somewhat similarly, but hearing what another says involves linguistic conventions and apparently relies on scaffolding that develops as part of our biological endowment; in addition, what is conveyed is not immediately veridical. In any case, the general point is that, with the appropriate scaffolding, ToM abilities provide a mechanism for achieving CSK. Indeed, ToM would be pointless if it did not do so and also provide access to the scaffolding.

5.8 Common State Knowledge and Shared Situation Awareness

The key feature of the participants that is established by TITAN disseminating information and that enables coordination is often identified as shared situation awareness (shared SA). Shared SA is typically characterized as, for example, "a reflection of how similarly team members view a given situation" (Bolstad, Cuevas, Gonzalez, & Schneider, 2005). Similarly, as it is seen, common ground (roughly what is shared in shared SA) is characterized as a shared perspective or shared knowledge, beliefs, and assumptions.

As noted, what is ignored is the simple but critical self-referential nature of the system, which is the foundation for the CSK that enables coordination. Here the account places the emphasis on the participants, their coordination, and their communication; perception is de-

emphasized. Although mental models for individuals are seen as especially useful in SA in general (Endsley, 1988) for understanding how information is integrated and the future is projected, the subject of shared SA do not typically connect one mental model with another. As seen, however, ToM suggests that our understanding of the beliefs and intentions of others is more natural than is often thought, and we suggest that this understanding is more accurate than is often thought if we share common social scaffolding.

The emphasis SA puts on projecting into the future (Endsley, 1988) is accepted, and its agreed that, in designing a system, how well the design "supports the operator's ability to get the needed information under dynamic operational constraints" (Endsley, 2000) should be addressed. From TITAN's point of view, the threat is not so much being overwhelmed by information about the physical environment as being overwhelmed by messages from collaborators. A military hierarchy is part of the solution, and scripting is another since it allows only relevant information to be passed up, down, and over in the hierarchy to keep the proxies consistent and synchronized.

5.9 Advantages of the Paradigms and their Integration

This section describes the advantages of different paradigms used for the implementation of TITAN services and the advantages of their integration. The advantages of multiagent systems include ease of modification or extension and high fault tolerance. Other advantages include the agents' resource planning and problem solving skills, where expertise and data are spatially or temporally distributed, and their ability to progress with incomplete or conflicting information (Wooldridge, 2009).

Having numerous agents and different types of agents allows flexibility (being proactive or reactive and being able to switch these two kinds of behavior) in tailoring a solution to the problem even if the problem changes with each new occurrence or changes while it is being

solved (Wooldridge, 2009). This is a major advance over the way most computer systems currently operate, where every possibility must be anticipated and coded for in advance. This social ability leads to a more robust system.

WADE adds to JADE functionality for the execution of the tasks defined according to the workflow metaphor and a number of management mechanisms. The main advantage of WADE is that it provides a direct way to implement workflows.

Web services (W3 Committee) are widely used in the military and other domains where definite information may need to be available publicly. They allow cooperation, communication, and integration on a wide scale. The potential of Web Services technology is enormous because, under the Web services paradigm, a single application can tap into the services of millions of applications that are scattered throughout the Internet.

ACL (agent communication language) is used for communication between agents present in distributed environments. ACL provides a declarative language to describe a desired state and defines the types and semantics of messages that agents are able to exchange (Labrou, Finin, & Yun, 1999). The Java Message Service (JMS) is an abstraction of the interfaces and classes needed by messaging clients when communicating with messaging systems. Integration of these provides functionality where agents can register and publish alert message services. JMS is a light weight protocol based on push technology which allows sending alerts and warnings for the agents that are subscribed to it.

Although Web services enable the execution of remote functions and messaging, they do not provide a robust infrastructure for handling information. Web services when combined with JMS and multiagent systems provide a reliable, scalable, and loosely coupled architecture for messaging. The combination of Web services with JMS creates an architecture that can

communicate efficiently across the Internet, reliably handle data, and integrate with backend systems.

Multiagent systems, Web services , distributed event-based systems (in the form of JMS) and workflows(in form of WADE) with the agents dominating the overall control flow, integrating JMS with the multiagent system provides greater message capability and also decouples the entire system. The incorporation of Web services allows information to be available publicly. Overall, the agents provide intelligence to the TITAN system and greatly enhance the capabilities of the other technologies.

CHAPTER 6

Conclusion and Future work

The Communications-Electronics Research, Development, and Engineering Center (CERDEC) Command and Control Directorate (C2D) manage the Information Dissemination and Management (ID&M) research and development work. The ID&M services are required to use intelligent-agent software frameworks because the tactical command and control domain is suitable for the application of this technology. The primary purpose of intelligent agents is to provide assistance to humans by acting independently on their behalf. On the battlefield, this assistance would be useful when it comes to tasks like pointing out when a plan is no longer feasible, which may not be immediately realized by war-fighters engaged in battle.

Several widespread issues will be addressed by the proposed research. There is the issue of integrating the four substantial technologies, namely, event-based systems, multi-agent systems, Web services and workflows. Presently, we have technology at hand to facilitate the integration of multi-agent systems and Web services (in the form of WSIG) and the integration of event-based systems and multiagent systems (in the form of the JMS-Agent Gateway). The proposed research will provide a context in which all aspects of these integration technologies can be thoroughly tested, both individually and in combination. This context will also have Web services and an event-based system working together through the agent technology. The four major technologies in this research provide ways for multiple entities to communicate and to collaborate.

The main goal of TITAN is to coordinate units in a military hierarchy by coordinating their TBSs (Titan Battle Command Supports), and a prerequisite for coordination is common knowledge among the units, which may arise in several ways. In all cases, a nested self-reference

within the group is critical. Common knowledge in a situation (common state knowledge, CSK) can arise under certain physical or linguistic co-presence conditions. There is also common knowledge by virtue of a mutual sharing of social scaffolding. In TITAN, the protocols designed into the TBSs provide scaffolding. Given an OPORD as a coordination script, JMS messaging maintains CSK. Because of communication constraints, the workflow that directs the child TBSs uses proxy agents to represent the children. This implements a version of the concept of theory of mind (ToM). With the appropriate scaffolding, ToM provides a mechanism for achieving CSK.

Future work will include implementing the rest of TITAN services described in the TITAN suite. It would also include using technologies that standardize asynchronous push abilities of Web service like WS-Eventing and WS-Notification. Since agents support flexible sequences of conversation and communication, there is motivation to try to enlist functionalities of the other two technologies under multiagent systems.

It is proposed here to use predefined workflow agents, which are invoked in the appropriate circumstances. Future work would include defining workflow agents dynamically to fit the precise circumstances at hand.

Hierarchies can be composed and decomposed in quite obvious and straightforward was (which is in fact a principal appeal of holons). In the future, we shall investigate how common knowledge is aggregated or split off when the sorts of military hierarchies envisioned by TITAN are composed or decomposed. It appears that scaffolding and CSK behave quite differently under these operations.

Future work will also address how much theory of mind is required of the various TBSs to maintain the required CSK. We expect some parent-child asymmetry in these requirements.

An additional issue is the extent to which the theory of mind attributed to a TBS can actually be attributed to the commander of the corresponding unit or even to the unit itself.

References

AOS Group. Report Agent Oriented Software Group JACK Retrieved April 15, 2010, from http://www.aosgrp.com/downloads/JACK_WhitePaper_US.pdf.

Armstrong, J., Virding, R., Wikström, C., & Williams, M. (1996). *Concurrent Programming in Erlang, Second Edition*. Hertfordshire,UK: Prentice-Hall.

Astington, J. W., & Jenkins, J. M. (1995). Theory of mind development and social understanding. *Cognition & Emotion, 9*(2), 151-165.

Babiceanu, R., & Chen, F. (2006). Development and Applications of Holonic Manufacturing Systems: A Survey. *Journal of Intelligent Manufacturing, 17*(1), 111-131.

Barker, A., & Hemert, J. V. (2008). Scientific workflow: a survey and research directions *Proceedings of the 7th international conference on Parallel processing and applied mathematics* (pp. 746-753). Gdansk, Poland: Springer-Verlag.

Barnette, M. (2008). *A Prototype Intelligent Multiagent Benchmark Maintaining Distributed Situational Awareness with a Rule Engine.* MS Thesis, North Carolina A&T State University, Greensboro, NC.

Barwise, J. (1989). On the Model Theory of Common Knowledge *The Situation in Logic* (pp. 201-220): Stanford, CA: Center for the Study of Language and Information.

Bellifemine, F. L., Caire, G., & Greenwood, D. (2007). *Developing multi-agent systems with JADE*. Hoboken, NJ: John Wiley & Sons.

Bolstad, C. A., Cuevas, H. M., Gonzalez, C., & Schneider, M. (2005). Modeling shared situation awareness *14th Conference on Behavior Representation In Modeling and Simulation BRIMS*.

Bratman, M. E. (1990). What is Intention. In P. R. Cohen, J. L. Morgan & M. E. Pollack (Eds.), *Intentions in communication*. Cambridge, MA: MIT Press.

Bratman, M. E., Israel, D. J., & Pollack, M. E. (1987). *Intention, Plans and Practical Reason*. Cambridge, MA: Harvard University Press.

Bratman, M. E., Israel, D. J., & Pollack, M. E. (1988). Plans and resource-bounded practical reasoning. *Computational Intelligence, 4*(3), 349-355.

Carzaniga, A., & Fenkam, P. (2004). *Third International Workshop on Distributed Event-Based Systems.* Paper presented at the Proceedings of the 26th International Conference on Software Engineering.

CERDEC US Army. (December 22, 2008). *OPORD Support Service Agent Architecture Specification*. (Version 1.5).

CERDEC US Army. (November 3, 2008). *Battle Book Service Agent Architecture Specification*. (Revision 1.1).

CERDEC US Army. (November 12, 2008a). *Alert and Warning Service Agent Architecture Specification.* (Revision 1.2).

CERDEC US Army. (November 12, 2008b). *Smart Filtering Assistance Agent Architecture Specification*. (Revision 1.2).

CERDEC US Army. (October 21, 2008). *Workflow Orchestration Support Service Agent Architecture Specification*. (Revision 1.0).

Chwe, M. S. Y. (2001). *Rational Ritual: Culture, Co-ordination and Common Knowledge*. Princeton,NJ: Princeton University Press.

Clark, A. (1998). *Being There: Putting Brain, Body, and World Together Again*: The MIT Press.

Clark, H. H. (1996). *Using Language*. Cambridge, UK: Cambridge University Press.

Clark, H. H., & Brennan, S. A. (1991). Grounding in communication. In L. B. Resnick, J. M. Levine & S. D. Teasley (Eds.), *Perspectives on socially shared cognition* (pp. 127-149).

Clark, H. H., & Carlson, T. B. (1982). Speech Acts and Hearers' Beliefs In N. V. Smith (Ed.), *Mutual knowledge* (pp. 1-37). Newyork: Academic Press.

Cougaar Team. (2004). Cougaar Developers' Guide Retrieved January 25, 2011, from http://cougaar.org/twiki/pub/Main/CougaarDeveloperGuide/CDG_11_4.pdf.

Curry, E. (March 29, 2004). How to use the Java Message Service (JMS)-Agent Gateway Retrieved April 15, 2011, from http://ecrg.it.nuigalway.ie/jade/jmsagentgateway/

Curry, E., Chambers, D., & Lyons, G. (2004). *Enterprise service facilitation within agent environments.* Paper presented at the IASTED Conference on Software Engineering and Applications, Cambridge, MA.

Dastani, M. An Abstract Agent Programming Language Retrieved April 17, 2010, from http://www.cs.uu.nl/3apl/.

Dastani, M. A Practical Agent Programming Language Retrieved April 20, 2010, from http://apapl.sourceforge.net/.

Data Management Working Group. (December 13, 2007). *JC3IEDM Overview*. Greding, Germany.

Deelman, E., & Gill, Y. (2006). Workshop on the Challenges of Scientific Workflows: University of Southern California.

Endsley, M. R. (1988). *Design and evaluation for situation awareness enhancement.* Paper presented at the Proc. of the Human Factors Society 32nd Annual Meeting.

Endsley, M. R. (2000). Theoretical underpinnings of situation awareness: a critical review. In M. R. Endsley & D. J. Garland (Eds.), *Situation Awareness Analysis and Measurement*:

Lawrence Erlbaum Associates.

Erl, T. (2005). *Service-Oriented Architecture: Concepts, Technology & Design*: Prentice Hall.

Esterline, A., BouSaba, C., Pioro, B., & Homaifar, A. (2006). Hierarchies, Holons, and Agent Coordination. In M. Hinchey, P. Rago, J. Rash, C. Rouff, R. Sterritt & W. Truszkowski (Eds.), *Innovative Concepts for Autonomic and Agent-Based Systems* (Vol. 3825, pp. 210-221): Springer Berlin / Heidelberg.

Esterline, A., Wright, W., & Banda, S. (2011). A System Integrating Agent Services with JMS for Shared Battlefield Situation Awareness. In D. Dicheva, Z. Markov & E. Stefanova (Eds.), *Third International Conference on Software, Services and Semantic Technologies S3T 2011* (Vol. 101, pp. 81-88): Springer Berlin / Heidelberg.

Fagin, R., Halpern, J. Y., Moses, Y., & Vardi, M. Y. (1995). *Reasoning about Knowledge*. Cambridge, MA: MIT Press.

FIPA Board. The Foundation for Intelligent Physical Agents Retrieved May 17, 2010, from http://www.fipa.org/

FIPA Board. The Foundation for Intelligent Physical Agents. How to Become a FIPA Member. Retrieved May 18, 2010, from http://www.fipa.org/subgroups/.

Flavell, J. (2004). Theory-of-mind development: Retrospect and prospect. *Merrill-Palmer Quarterly*, *50*(3), 274-290.

Friedman-Hill, E. Jess: the Rule Engine for the Java Platform Retrieved June 05, 2010, from http://jessrules.com/.

Friedman-Hill, E. (2003). *Jess in Action*. Greenwich, CT: Manning Publications Company.

Goldman, A. I. (2006). *Simulating minds: the philosophy, psychology, and neuroscience of mindreading*: Oxford University Press.

Greenwood, D. (2005). *JADE Web Service Integration Gateway* Paper presented at the Autonomous Agents and Multiagent Systems, Utrecht, The Netherlands.

Haefel, R. M., & Chappell, D. A. (2001). *Java Message Service*: O'Reilly.

Haridi, S., & Franzen, N. Tutorial of Oz Retrieved May 02, 2011, from http://www.mozart-oz.org/documentation/tutorial/

Headquarters, D. O. A. (31 May, 1997). *Field Manual 101-5.* Washington D.C.

Headquarters, D. O. A. (August 11, 2003). *Field Manual 6.0, Chapter 5*. Washington D.C.

Hendricks, V., & Symons, J. (2009). Epistemic Logic. In E. N. Zalta (Ed.), *The Stanford Encyclopedia of Philosophy*.

Hewitt, C. (2007). What Is Commitment? Physical, Organizational, and Social (Revised). In N. Pablo, V. Javier, S. zquez, B. Guido, B. Olivier, D. Virginia, F. Nicoletta & M. Eric (Eds.), *Coordination, Organizations, Institutions, and Norms in Agent Systems II* (pp. 293-307): Springer-Verlag.

Hewitt, C., Bishop, P., & Steiger, R. (1973). *A universal modular ACTOR formalism for artificial intelligence*. Paper presented at the Proceedings of the 3rd international joint conference on Artificial intelligence, Stanford, USA.

Hilaire, V., Koukam, A., & Rodriguez, S. (2008). An adaptative agent architecture for holonic multi-agent systems. *ACM Trans. Auton. Adapt. Syst.*, *3*(1), 1-24.

Hübner, J. F., & Bordini, R. H. Jason Retrieved April 16, 2010, from http://jason.sourceforge.net/Jason/Jason.html.

. Introduction. (1996). In R. Bilger, S. Guest & M. J. Tauber (Eds.), *Electronic Proceedings Conference on Human Factors in Computing Systems*. Vancouver, BC.

JADE-Board. JADE:Java Agent Development Framework Retrieved May 15, 2010, from http://jade.tilab.com.

Kappler, M. (2009). Cougaar New User Road Map. Retrieved January 30 2011, from http://cougaar.org/twiki/bin/view/Main/CougaarRoadmap

Kitchin, D., Quark, A., Cook, W., & Misra, J. (2009). The Orc Programming Language Formal Techniques for Distributed Systems. In D. Lee, A. Lopes & A. Poetzsch-Heffter (Eds.), (Vol. 5522, pp. 1-25): Springer Berlin / Heidelberg.

Koestler, A. (1968). *The Ghost in the Machine*. Newyork: Macmillan.

Krishnamurthy, K. (2010). *A system that complements agent services with web services and an event-based system.* MS Thesis, North Carolina A&T State University, Greensboro, NC.

Labrou, Y., Finin, T., & Yun, P. (1999). Agent communication languages: the current landscape. *Intelligent Systems and their Applications, IEEE, 14*(2), 45-52.

Mason, J. (2008). *A Benchmark that Integrates Web Services into a Multiagent Distributed Situational Awareness System.* MS Thesis, North Carolina A&T State University, Greensboro, NC.

Milner, R. (1989). *Communication and Concurrency*. Englewood Cliffs, NJ: Prentice-Hall.

Misra , J. (2004). Computation orchestration: A basis for wide-area computing. In M. Broy (Ed.), *NATO Advanced Study Institute, Engineering Theories of Software Intensive Systems, NATO ASI Series*. Marktoberdorf, Germany.

Mühl, G., Fiege, L., & Pietzuch, P. (2006). *Distributed Event-Based Systems*. Berlin: Springer.

Multi agent Group. Janus Project Retrieved March 28, 2011, from http://www.janus-project.org/Home

Myers, K. L. PRS-CL: A Procedural Reasoning System Retrieved April 16, 2010, from http://www.ai.sri.com/~prs/

Nofi, A. A. (2002). Defining and Measuring Shared Situational Awareness. Retrieved from http://www.cna.org/documents/D0002895.A1.pdf

OASIS WSBPEL Technical Committee. (2007). Web Services Business Process Execution Language Retrieved May 05, 2011, from http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html.

Ort, E. Java Architecture for XML Binding Retrieved August 30, 2010, from http://www.oracle.com/technetwork/articles/javase/index-140168.html.

Pérez-Quiñones, M. A., & Sibert, J. L. (1996). *A collaborative model of feedback in human-computer interaction*. Paper presented at the Proceedings of the SIGCHI conference on Human factors in computing systems: common ground, Vancouver, British Columbia, Canada.

Pierce, B. C., & Turner, D. N. (2000). Pict: a programming language based on the Pi-Calculus. In G. Plotkin, C. Stirling & M. Tofte (Eds.), *Proof, language, and interaction: Essay in Honour of Robin Milner* (pp. 455-494): MIT Press.

Process Mining Board. Process Mining Retrieved May 10, 2011, from http://www.processmining.org/.

process Mining Board. Process Mining: Prom Retrieved May 15, 2011, from http://www.processmining.org/prom/.

Reid, D. (2007). *Agent Prototype for a Distributed Situational Awareness System.* MS Thesis, North Carolina A&T State University, Greensboro, NC.

Schillo, M., & Fischer, K. (2003). Holonic Multiagent Systems. *KI*, *4*(54), 54-55.

Singh, M., & Huhns, M. (2005). *Service-Oriented Computing: Semantics, Processes, and Agents*: John Wiley & Sons.

Sjöberg, P. (2007). The Java Message Service 1.0.2 Retrieved August 10, 2010, from http://www.cs.helsinki.fi/u/campa/teaching/j2me/pa

SPARK Team. SPARK Retrieved April 16, 2010, from http://www.ai.sri.com/~spark.

The Apache Software Foundation. jUDDI: An Apache Project Retrieved July 23, 2010, from http://ws.apache.org/juddi/.

The Apache Software Foundation. Welcome to Apache Axis2/Java Retrieved July 21, 2010, from http://axis.apache.org/axis2/java/core/.

Tomasello, M. (2009). *Why We Cooperate* Cambridge, MA: The MIT Press.

UDDI Spec Technical Committee. OASIS: UDDI Version 3.0.2 Retrieved July 21, 2010, from http://www.uddi.org/pubs/uddi_v3.htm.

US Army. Defense Advanced Research Projects Agency Retrieved January 30, 2011, from http://www.darpa.mil/.

US Army. (2008). *A Guide for Developing Agent-Based Services*. (14.0).

Van der Aalst, W. M. P., Dumas, M., Ter Hofstede, A. H. M., Russell, N., A Wohed, P., & A Verbeek, H. M. W. (2005). *Life After BPEL?* Paper presented at the International Workshop on Web Services and Formal Methods (WS-FM).

Van der Aalst, W. M. P., & Hee, K. V. (2002). *Workflow Management: Model, Methods & Systems*. Cambridge, MA: MIT Press.

Van der Aalst, W. M. P., Reijers, H. A., Weijters, A. J. M. M., Van dongen, B. F., Alves de Mederios, A. K., Song, M., & Verbeek, H. M. W. (2007). Business process mining: An industrial application. *Inf. Syst.*, *32*(5), 713-732.

Van der Aalst, W. M. P., & ter Hofstede, A. H. M. (2005). YAWL: yet another workflow language. *Information Systems, 30*(4), 245-275.

Vermeersch, R. (2009). Concurrency in Erlang & Scala: The Actor Model Retrieved March 15, 2011, from http://ruben.savanne.be/articles/concurrency-in-erlang-scala

W3 Committee. W3C: World Wide Web Consortium Retrieved July 20, 2010, from http://www.w3.org/.

W3C Team. Simple Object Access Protocol (SOAP) Retrieved July 10, 2010, from http://www.w3.org/TR/2000/NOTE-SOAP-20000508/

WADE Board. WADE: Workflow Agents Development Environment Retrieved January 20, 2011, from http://jade.tilab.com/wade/.

Weiss, G. (1999). *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*. London, England: MIT Press Cambridge.

Wiriyacoonkasem, S., & Esterline, A. C. (2001). *Heuristics for Inferring Common Knowledge via Agents' Perceptions in Multiagent Systems.* Paper presented at the Proceedings of the 5th World Multi-conference on Systemics, Cybernetics, and Informatics Conference, Orlando, FL.

Wooldridge, M. (2009). *An Introduction to MultiAgent Systems*: John Wiley & Sons.

YAWL Team. YAWL: Yet Another Workflow Language. Retrieved May 15, 2011, from http://yawlfoundation.org/.

Appendix A

Monitor Agent

The Monitor Agent publishes and subscribes to messages and also stores archive of messages which are executed by the agents in the WOS Service.

```
import ie.nuigalway.ecrg.jade.jmsagentgateway.Util;
import ie.nuigalway.ecrg.jade.jmsagentgateway.onto.JmsAgentGatewayOntology;
import ie.nuigalway.ecrg.jade.jmsagentgateway.onto.JmsMessage;
import ie.nuigalway.ecrg.jade.jmsagentgateway.onto.OnMessage;
import ie.nuigalway.ecrg.jade.jmsagentgateway.onto.PublishMessage;
import ie.nuigalway.ecrg.jade.jmsagentgateway.onto.Subscribe;
import java.io.File;
import java.io.FileInputStream;
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import jade.content.ContentElement;
import jade.content.lang.Codec;
import jade.content.lang.sl.SLCodec;
import jade.content.onto.Ontology;
import jade.content.onto.basic.Action;
import jade.content.AgentAction;
import jade.core.AID;
```

```
import jade.core.Agent;

import jade.core.behaviours.OneShotBehaviour;

import jade.domain.DFService;

import jade.domain.FIPAAgentManagement.DFAgentDescription;

import jade.domain.FIPAAgentManagement.ServiceDescription;

import jade.lang.acl.ACLMessage;

import jade.proto.SubscriptionInitiator;

import javax.xml.bind.JAXBContext;

import javax.xml.bind.JAXBElement;

import javax.xml.bind.JAXBException;

import javax.xml.bind.Marshaller;

import javax.xml.bind.Unmarshaller;

import java.util.ArrayList;

import java.util.List;

import java.util.*;

import army.*;

public class MonitorAgent extends Agent {

private AID jmsProxyAgent;

private int ctr = 0;

private Codec codec = new SLCodec();

private Ontology ontology = JmsAgentGatewayOntology.getInstance();

public String payload_recieved = new String("");

public Agent Myagent = this;
```

```
public WebTestClient w1;
  //  Setup the agent
     protected void setup() {
  // Register language and ontology
    getContentManager().registerLanguage(codec);
    getContentManager().registerOntology(ontology);
    lookupAgent();
            // Set this agent main behaviour
    addBehaviour(new JmsSubscriptionInitiator(this, getSubscriptionMsg("topicD")));
  }
//    Utility message to publish a message
 //   @param payload  Payload of the message to be published
     private void publishMessage(String payload,String topic_name) {
    PublishMessage pm = new PublishMessage();
    try {
      pm.setMessage(Util.createMessage(topic_name, payload, Util.PERSISTENT));
    } catch (Exception e) {
      System.out.println("Error creating message:" + e.toString());
    }
    // Using a connection with the JBoss provider
    pm.setProviderInfo(Util.createProviderInfo(Util.TOPIC, "jnp://localhost:1199",
            "org.jnp.interfaces.NamingContextFactory"));
    System.out.println("Sending message: " + payload);
```

```
    sendMessage(ACLMessage.REQUEST, pm);
  }
// Utility method for sending a message
 //  @param performative  Performative for the message
  //@param  action        The AgentAction of the message to be sent
  private void sendMessage(int performative, AgentAction action) {
    ACLMessage msg = new ACLMessage(performative);
    msg.setLanguage(codec.getName());
    msg.setOntology(ontology.getName());
    try {
      getContentManager().fillContent(msg, new Action(jmsProxyAgent, action));
      msg.addReceiver(jmsProxyAgent);
      send(msg);
    } catch (Exception e) {
      System.out.println("Error sending message:" + e.toString());
      e.printStackTrace();
    }
  }
// Lookup the JmsProxyAgent
    private void lookupAgent() {
 // Publisher Agent
    ServiceDescription sd = new ServiceDescription();
    sd.setType("JmsProxyAgent");
```

```
    DFAgentDescription dfd = new DFAgentDescription();
    dfd.addServices(sd);
    try {
      DFAgentDescription[] dfds = DFService.search(this, dfd);
      if (dfds.length > 0) {
        jmsProxyAgent = dfds[0].getName();
      } else {
        System.out.println("Couldn't localize server!");
      }
    } catch (Exception e) {
      e.printStackTrace();
      System.out.println("Failed searching int the DF!");
    }
  }
// Create a subscription message
// @return    The subscriptionMsg value
    private ACLMessage getSubscriptionMsg(String name) {
                Subscribe sub = new Subscribe();
                sub.setSubscription(Util.createSubscription(name, Util.NON_DURABLE));
   // Using a connection with the JBoss provider
                sub.setProviderInfo(Util.createProviderInfo(Util.TOPIC, "jnp://localhost:1099",
            "org.jnp.interfaces.NamingContextFactory"));
                ACLMessage msg = new ACLMessage(ACLMessage.SUBSCRIBE);
```

```
            msg.setLanguage(codec.getName());
            msg.setOntology(ontology.getName());
            try {
              getContentManager().fillContent(msg, new Action(jmsProxyAgent, sub));
              msg.addReceiver(jmsProxyAgent);
            } catch (Exception e) {
              System.out.println("Error populating subscription message:" + e.toString());
              e.printStackTrace();
            }
            return msg;
          }
//  Implementation of the SubscriptionInitiator interface
        class JmsSubscriptionInitiator extends SubscriptionInitiator {
            private int counter = 0;
            Agent agent;
//  Constructor for the JmsSubscriptionInitiator object
// @param  a    Agent involved in this interaction
//@param  msg  Subscription message to send
              JmsSubscriptionInitiator(Agent a, ACLMessage msg) {
              super(a, msg);
              agent = a;
            }
//Method called when a Agree message is received
```

```
// @param  agree  Agree message received
        protected void handleAgree(ACLMessage agree) {
          System.out.println("Handle Agree");
        }
// Method called when a Refuse message is received
// @param  refuse  Refuse message received
          protected void handleRefuse(ACLMessage refuse) {
          System.out.println("Handle Refuse");
        }
// Method called when a NotUnderstood message is recceived
// @param  notUnderstood  NotUnderstood message received
        protected void handleNotUnderstood(ACLMessage notUnderstood) {
          System.out.println("Handle not understood");
        }
//  Method called when a Inform message is received
// @param  inform  Inform message received
          protected void handleInform(ACLMessage inform) {
          System.out.println("Handle Inform");
          try {
            ContentElement content = getContentManager().extractContent(inform);
            OnMessage om = (OnMessage) ((Action) content).getAction();
            JmsMessage jmsMsg = om.getMessage();
```

```
                    payload_recieved =
(String)Util.convertStringToObject(jmsMsg.getPayload());
        System.out.println("Message Received: Payload:" + ((String)
Util.convertStringToObject(jmsMsg.getPayload())));
                    w1 = new WebTestClient();
                    String soap_response = new String(w1.ret_response());
                    Integer val = new Integer(payload_recieved);
                System.out.println("This is Topic D\n");
            //Publish message to topicE
            if(soap_response.contains("RED"))
            {
                    System.out.println("Udpating the OPORD");
                    update_task(val.intValue());
                    addBehaviour(new PublishCommand(Myagent,"topicE"));
            }
            else if(soap_response.contains("BLUE"))
            {
                    Random generator = new Random();
                    int r = generator.nextInt();
                    if((r%2)==0)
                    {
                            System.out.println("Udpating the OPORD");
                            update_task(val.intValue());
```

```
                    addBehaviour(new PublishCommand(Myagent,"topicE"));
                }
                else
                {
                    System.out.println("No udpates to be done");
                }
            }
            else if (soap_response.contains("GREEN"))
            {
                System.out.println("No udpates to be done");
            }
        }catch (JAXBException je) {
    je.printStackTrace();
  } catch (IOException ioe) {
    ioe.printStackTrace();
  } catch (Exception e) {
        e.printStackTrace();
      }
    }
//  Method called when a Failure message is received
// @param  failure  Failure message received
    protected void handleFailure(ACLMessage failure) {
      System.out.println("Handle Failure");
```

```
            }
        }
  // Behaviour used to publish messages to the JmsProxyAgent
  class PublishCommand extends OneShotBehaviour {
    private String Topic_name;
  //   Constructor for the PublishCommand object
// @param  a  Agent publishing messages
    PublishCommand(Agent a, String name) {
      super(a);
            this.Topic_name = new String(name);
    }
//  Action of the behaviour
    public void action() {
          publishMessage(payload_recieved,Topic_name);
    }
  }
```

Appendix B

Orchestration Agent

The Orchestration Agent receives the OPORD and reads it and initiates and traces the Workflow Agent to execute workflows.

```
import java.util.HashMap;

import java.util.Map;

import com.tilab.wade.commons.WadeAgent;

import com.tilab.wade.dispatcher.DefaultEventListener;

import com.tilab.wade.dispatcher.DispatchingCapabilities;

import com.tilab.wade.dispatcher.WorkflowResultListener;

import com.tilab.wade.performer.Constants;

import com.tilab.wade.performer.descriptors.ElementDescriptor;

import com.tilab.wade.performer.descriptors.WorkflowDescriptor;

import com.tilab.wade.performer.event.BeginActivity;

import com.tilab.wade.performer.event.BeginWorkflow;

import com.tilab.wade.performer.event.EndActivity;

import com.tilab.wade.performer.event.EndWorkflow;

import com.tilab.wade.performer.ontology.ControlInfo;

import com.tilab.wade.performer.ontology.ExecutionError;

import com.tilab.wade.utils.DFUtils;

import jade.core.AID;

import jade.core.Agent;

import jade.util.leap.ArrayList;
```

```
import jade.util.leap.List;
public class OrchestrationAgent extends Agent {
	private DispatchingCapabilities dc;
	protected void setup() {
	dc  = new DispatchingCapabilities();
		dc.init(this);
		try {
		// Search for an agent able to execute workflows
		AID executor = DFUtils.getAID(DFUtils.searchAnyByRole(this,
	WadeAgent.WORKFLOW_EXECUTOR_ROLE, null));
			while (executor == null) {
// Possibly the executor has not registered with the DF yet --> Wait a bit then try again
				Thread.sleep(1000);
				executor = DFUtils.getAID(DFUtils.searchAnyByRole(this,
	WadeAgent.WORKFLOW_EXECUTOR_ROLE, null));
			}
			System.out.println("Executor found: "+executor.getLocalName());
// Create the WorkflowDescriptor specifying the workflow to execute and the parameters
			WorkflowDescriptor wd = new WorkflowDescriptor("WorkflowAgent");
			Map<String, Object> parameters = new HashMap<String, Object>();
			parameters.put("message", "Hello World!");
			wd.setParametersMap(parameters);
```

```
// Prepare the control-information to make the workflow generate and send me FLOW events at
        // the ACTIVITY level
                List infos = new ArrayList();
                infos.add(new ControlInfo(Constants.FLOW_TYPE, getAID(),
        Constants.ACTIVITY_LEVEL));
// Launch the workflow
dc.launchWorkflow(executor, wd, new WorkflowResultListener() {
        public void handleAssignedId(AID executor, String executionId) {
                // The executor accepted the execution of the workflow.
                // Register a listener for FLOW events
                dc.setEventListener(new FlowEventListener(), Constants.FLOW_TYPE,
        executor, executionId);
                        }
        public void handleExecutionCompleted(List results, AID executor, String executionId) {
                // Workflow execution successfully completed
                System.out.println("Workflow completed successfully");
                // Get output parameters
        Map<String, Object> params = ElementDescriptor.paramListToMap(results);
                        int length = (Integer) params.get("messageLength");
                        System.out.println("Output parameter messageLength =
        "+length);
        }
public void handleExecutionError(ExecutionError er, AID executor, String executionId) {
```

```
                // Workflow execution failed
        }
public void handleLoadError(String reason) {
                // Executor failed/refused to load the workflow
                                }
public void handleNotificationError(AID executor, String executionId) {
                // Generic communication error. This should never happen
                                }
                }, infos);
                }
catch (Exception e) {
e.printStackTrace();
                }
        }
private class FlowEventListener extends DefaultEventListener {
                public void handleBeginWorkflow(long time, BeginWorkflow event, AID
        executor, String executionId) {
                        System.out.println(">>>> Begin-workflow "+event.getName());
                }
public void handleEndWorkflow(long time, EndWorkflow event, AID executor, String
        executionId) {
                        System.out.println(">>>> End-workflow "+event.getName());
                }
```

```
public void handleBeginActivity(long time, BeginActivity event, AID executor, String
        executionId) {
                System.out.println(">>>> Begin-activity "+event.getName());
            }
public void handleEndActivity(long time, EndActivity event, AID executor, String executionId)
        {
                System.out.println(">>>> End-activity "+event.getName());
            }
        }
public void update_task(int in)
        {
            try{
            JAXBContext jc = JAXBContext.newInstance("army");
// create an Unmarshaller
      Unmarshaller u = jc.createUnmarshaller();
            String fname =
        "C:/Users/ats/Desktop/SrinivasCode/prdC2ICS0_MLS1_7DIVtemplate&text&MsnTOint
        OpsExe_100119.xml";
        PrdC2 p1 = (PrdC2)u.unmarshal(new FileInputStream(fname));
        TpcExe Te1 = p1.getTpcExe();
        List<Fct> Fct1 = new ArrayList(Te1.getFct());
        List<Fct> Fct11 = new ArrayList();
        List<Unit> unit1 = new ArrayList();
```

```
for(int i =0; i< Fct1.size();i++)
{
   if(Fct1.get(i).getType() == FctTypeST.TSK)
   {
      Fct11.addAll(Fct1.get(i).getFct());
      System.out.println(Fct1.get(i).getType());
      break;
   }
}
for(int i =0; i< Fct11.size();i++)
{
   unit1.addAll(Fct11.get(i).getUnit());
}
if(in < unit1.size()){
         String name = new String(unit1.get(in).getDsg());
         for(int i = in; i < unit1.size();i++)
         {
                  if(unit1.get(in).getDsg().contentEquals(name))
                           unit1.get(in).setDsg("MY UNIT - "+ ctr);
         }
         ctr++;
         File abc = new File(fname);
         Marshaller m = jc.createMarshaller();
```

```
                    m.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT,
    Boolean.TRUE);
                    m.marshal(p1, abc );
            }
            }catch (JAXBException je) {
    je.printStackTrace();
  } catch (IOException ioe) {
    ioe.printStackTrace();
  }
}
}
```

Appendix C

Workflow Agent

The Workflow Agent is an agent that executes the main workflow. It creates two subflows which are executed concurrently. Code is presented below.

```
import com.tilab.wade.performer.layout.TransitionLayout;
import com.tilab.wade.performer.Transition;
import com.tilab.wade.performer.layout.MarkerLayout;
import com.tilab.wade.performer.layout.ActivityLayout;
import com.tilab.wade.performer.layout.WorkflowLayout;
import com.tilab.wade.performer.CodeExecutionBehaviour;
import com.tilab.wade.performer.WorkflowBehaviour;
import com.tilab.wade.performer.FormalParameter;
@WorkflowLayout(entryPoint = @MarkerLayout(position = "(481,15)", activityName =
        "Initialize"),
        exitPoints = { @MarkerLayout(position = "(482,521)", activityName = "Final") },
        transitions = {@TransitionLayout(to = "Final", from = "subflow2"),
        @TransitionLayout(to = "Subflow2", from = "Initialize"), @TransitionLayout(to =
        "Final", from = "Subflow1"),
         @TransitionLayout(to = "Subflow1", from = "Initialize") }, activities = {
        @ActivityLayout(label = "Subflow2", position = "(588,234)", name = "Subflow2"),
        @ActivityLayout(label = "Final", position = "(451,394)", name = "Final"),
        @ActivityLayout(label = "Subflow1", position = "(312,241)", name = "Subflow1"),
        @ActivityLayout(label = "Initialize", position = "(450,91)", name = "Initialize") })
```

```
public class Workflowagent extends WorkflowBehaviour {
        public static final String SUBFLOW2_ACTIVITY = "Subflow2";
        public static final String FINAL_ACTIVITY = "Final";
        public static final String SUBFLOW1_ACTIVITY = "Subflow1";
        public static final String INITIALIZE_ACTIVITY = "Initialize";
        @FormalParameter(mode=FormalParameter.INPUT)
        public String message;
        @FormalParameter(mode=FormalParameter.OUTPUT)
        public int messageLength;
        private void defineActivities() {
        CodeExecutionBehaviour initializeActivity = new CodeExecutionBehaviour(
                                                        INITIALIZE_ACTIVITY, this);
        registerActivity(initializeActivity, INITIAL);
        SubflowDelegationBehaviour Subflow1Activity = new SubflowDelegationBehaviour(
                                                        SUBFLOW1_ACTIVITY, this);
        Subflow1Activity.setSubflow("SubflowWorkflow1");
        registerActivity(Subflow1Activity);
        CodeExecutionBehaviour finalActivity = new CodeExecutionBehaviour(
                                                            FINAL_ACTIVITY, this);
        registerActivity(finalActivity, FINAL);
        SubflowDelegationBehaviour Subflow2Activity = new SubflowDelegationBehaviour(
                                SUBFLOW2_ACTIVITY, this);
        Subflow2Activity.setSubflow("SubflowWorkflow2");
```

```
        Subflow2Activity.setAsynch();
        registerActivity(Subflow2Activity);
        }
        private void defineTransitions() {
                registerTransition(new Transition(), INITIALIZE_ACTIVITY,
                                        SUBFLOW1_ACTIVITY);
                registerTransition(new Transition(), SUBFLOW1_ACTIVITY,
                                        FINAL_ACTIVITY);
                registerTransition(new Transition(), INITIALIZE_ACTIVITY,
                                        SUBFLOW2_ACTIVITY);
                registerTransition(new Transition(), SUBFLOW2_ACTIVITY,
                                        FINAL_ACTIVITY);
        }
        protected void executeInitialize() throws Exception {
                System.out.println("Initializing Workflow Agent!!!");
        }
protected void executeSubflow1(Subflow s) throws Exception {
                performSubflow(s);
        }
protected void executeFinal() throws Exception {
        }
protected void executeSubflow2(Subflow s) throws Exception {
                performSubflow(s);} }}
```