Prototype And Analysis Of The Army'S Titan Services

Mahesh Shivalingappa
North Carolina Agricultural and Technical State University

PROTOTYPE AND ANALYSIS OF THE ARMY'S TITAN SERVICES

by

Mahesh Shivalingappa

A thesis submitted to the graduate faculty
in partial fulfillment of the requirements for the degree of
MASTER OF SCIENCE

Department: Computer Science
Major: Computer Science
Major Professor: Dr. Albert C. Esterline

North Carolina A&T State University
Greensboro, North Carolina
2012

School of Graduate Studies
North Carolina Agricultural and Technical State University

This is to certify that the Master's Thesis of

Mahesh Shivalingappa

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

Greensboro, North Carolina
2012

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

Dedication

This thesis is dedicated to my parents: Shivalingappa Lingaiah and Ambuja Shivalingappa, and to my brother, Hitesh Shivalingappa.

Biographical Sketch

Mahesh Shivalingappa was born in Bangalore, India on March 30th, 1988. He received his Bachelor of Engineering degree in 2010 in the Department of Information Science and Engineering at the Vishveswaraiah Technological University, Bangalore, India. He joined North Carolina Agricultural and Technical State University in August 2010, where he is currently a candidate for Master of Science and a research assistant in the department of Computer Science and Engineering.

Acknowledgments

I would like to express the deepest appreciation to my advisor, Dr. Albert C. Esterline, who instilled in me, a spirit of imagination and 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 on the panel as committee members for my thesis defense, and providing me with valuable advice and guidance to make this dissertation possible. I would also like express deep appreciation and thanks to William Wright for providing invaluable insight into all the numerous discussions we undertook during the entire course of this research.

Last but not the least; I am very thankful to my family and friends for the unconditional love, support and encouragement they have constantly bestowed on me, enabling me to maintain the energy levels that was required to keep working on this thesis.

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

Shivalingappa, Mahesh. PROTOTYPE AND ANALYSIS OF THE ARMY'S TITAN SERVICES (**Major Professor: Dr. Albert C. Esterline**), North Carolina Agricultural and Technical State University.

The Tactical Information Technologies for Assured Net Operations (TITAN) Program is an Advanced Technology Objective (ATO) funded by the Army. The overall goal of this ATO is to demonstrate how emerging information technologies can significantly improve key areas of tactical operations, ultimately resulting in the transition 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 with Web Services and Java Messaging Services used for communication. Prototypes for three TITAN services namely, the Alert and Warning Service (AWS), the OPORD Support Service (OPS) and the Workflow Orchestration Support (WOS) service have all been implemented in an earlier phase of the project. The work reported here extends the functionality of the AWS service by implementing an Assessor Agent that provides a graphical interface, displaying the shortest/safe path through the area of operation of a phase of the TITAN mission. The work also discusses how the TITAN program established a Common knowledge for coordinating the hierarchically structured TITAN units. Common knowledge is modeled as a whole in Parts/whole Statecharts, simulated as a Stateflow.

CHAPTER 1

INTRODUCTION

The Tactical Information Technologies for Assured Net Operations (TITAN) Program [1] is an Advanced Technology Objective (ATO) funded by the Army. The overall goal of this ATO is to demonstrate how emerging information technologies can significantly improve key areas of tactical operations, ultimately resulting in the transition 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 used is JADE [2], whose agents envisioned for the TITAN Program here use Web Services [3] and Java Messaging services (in addition to agent communication supported by JADE) [4] for communication among them.

The implementation is built on top some work already done in the earlier phases of this project that prominently includes the realization of three of the several services defined under the ID&M portion of TITAN. Prototypes of the AWS, OPS and WOS services have been implemented earlier and this work extends the AWS service functionality. This however, is a small effort 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 larger effort aims at a complete implementation of the TITAN software suite using Cougaar [5] as the agent framework as opposed to JADE which is used in this project.

This project has resulted in the implementation of an extended Assessor Agent, which forms the heart of the AWS service and which was initially implemented as a very simple yet functional stub during an earlier phase of this project. The Assessor Agent thus extended provides for a fully informative graphical interface, mapping the shortest and more importantly, the safest paths through an area of operation during a particular phase of the TITAN Mission. The assessor agent is implemented as a dynamic algorithm incorporating a shortest path algorithm (to find the shortest path across an area of operation of a phase) and a convex hull algorithm (to safely navigate through regions densely populated with concealed explosive devices such as IEDs, etc.).

This project also discusses the coordination framework across the multi- echelon organization of the TITAN mission hierarchy. Common knowledge is considered as the pre-requisite for coordination and we have analyzed this with the help of Parts/Whole Statecharts implemented as a Stateflow.

The remainder of this thesis is organized as follows. The next chapter discusses the underlying technologies for the work needed to understand the implementation of the work in this project. The third chapter discusses the TITAN program. The fourth chapter discusses about the architecture and implementation of the work. The penultimate chapter discusses the Holorchical modeling of the TITAN platoon mission using the Parts/Whole Statecharts as a modeling technique realized using Stateflow. The seventh chapter concludes and discusses possible future work to extend the work done here.

CHAPTER 2

BACKGROUND

This chapter presents background information essential for understanding the work being proposed. The first section below presents distributed event-based systems in general, and the following section addresses the Java Messaging Service (JMS) in particular. Section 2.3 discusses Web services, and the next section introduces XML: all data communicated among TITAN services is in the form of XML documents conforming to a single XML schema. The third and last paradigm incorporated into the TITAN project – Multiagent Systems is covered in Section 2.6. The next section introduces JADE which complies with FIPA standards – FIPA us an IEEE standards organization for agent technology. The section following this, talks about how intelligence is incorporated into the JADE agents with the help of JESS. Section 2.8 covers WSIG (which makes JADE agent services available as Web services) and JMS-Agent Gateway, (which makes JMS functionality available to JADE agents in an agent-oriented way). The section that follows, addresses Workflow and Agents Development Environment (WADE) which adds to JADE functionality for execution of tasks that are defined as workflows. Section 2.10 introduces the concepts of common knowledge and theory of mind as pre-requisites for co-ordination. The final two sections of this chapter introduce the holon concept and how the holon can be modeled using Parts/Whole Statechart (which is a UML standard for modeling compound and reactive systems) and see use the Stateflow toolbox to realize the Parts/Whole model obtained.

2.1 Distributed Event based system

Systems composed of reactive components that have the ability to co-operate by exchanging information and control in the form of events [6], are collectively termed as event-based systems. In a distributed event-based system, the components are physically distributed over a network and the events are propagated to these remote components [7], which communicate with each other by generating and receiving event notifications. An *event* here is any transient occurrence of a happening of interest (e.g. a state change in some component) and a *notification* is a data representation that describes such an occurrence [8]. Different notifications can be created to describe the same event but from multiple viewpoints, and the reason for this is either due to application, security reasons or simply because the notifications are encoded in different data models. The most common of these data models are name/value pairs, objects and semi-structured data (e.g. XML) [6].

The notifications are conveyed via messages which are simply containers that transmit data on any underlying communication medium. A component usually generates an event notification when it wants to let the "external world" know of some relevant event that has occurred either in its internal state or in the state of another component with which it interacts. When such an event notification is generated, it is propagated to any component that has declared interest in receiving it and this generation and propagation of event notifications happens asynchronously [9].

The operation and practical description of an event-based style can be depicted by understanding the pictorial representation of its architecture in Figure 2.1. The architecture here is characterized by 2 main entities, the first of which is a connector by name *event notification service* (or just event service), responsible for dispatching event notifications [9] and secondly, *components* that collectively cooperate with each other through the event service. The components are further classified as *recipients* and *objects of interest*.

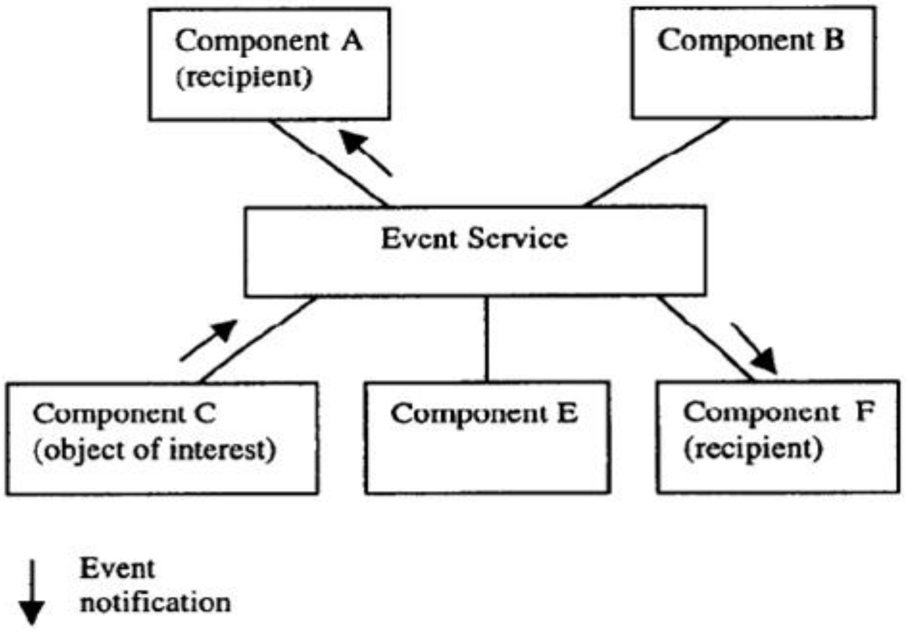

Figure 2.1. Architecture of an event-based system [9]

The objects of interest (also referred to as *producers*) are the components that publish notifications and whose implementations are self-focused in the sense, they observe only their own states. The producer notifies the occurrence of an event by first sending a publish request to the event service. Note that this notification is not addressed to any specific recipient; instead they are simply forwarded to the event service to be further distributed. The event service on receiving the publish request forwards the corresponding notification to all recipients who have subscribed for it. The recipients (also referred to as *consumers*) on the other hand declare their interest in receiving an event notification by subscribing for the particular event with the event service. Consumers, just like the producers are unaware of their communication peers and hence send out subscriptions to the event service describing the kinds of notifications they are interested in, and are not really concerned about the corresponding producers of the notifications. A component can behave both as an object of interest and as a recipient of events. If it behaves as both, it reacts to both internal notifications and observed internal state changes, and the resulting computations may lead to newly published notifications.

Consumers express their interest in receiving a particular kind of notification by issuing *subscriptions* to the notification service. The service in-turn evaluates the subscriptions on behalf of the consumer and delivers those notifications that match with the consumers' subscriptions. Subscriptions are analogous to *filters*, which are basically Boolean-valued functions that test a single notification and returns either *true* or *false*. They also include (Meta) data to govern the notification selection beyond a per-notification level. Also, a very important functionality of the *event notification service* is that it acts as a mediator in the event-based system which helps in decoupling the producers from the consumers and this functionality has two important effects. Firstly, a component can operate in the system without being aware of the existence of other components. All it has to know is the structure of the event notifications that are interesting to it and secondly, it is always possible to plug a component in and out of the architecture without directly affecting the other components. This loose coupling of producers and consumers in an event-based system leads to a flexible and robust system design in the Information Dissemination and Management applications.

2.2 Java Message Service

The event-based system is implemented here with the help of Java message service which is an API specification for enterprise messaging created by Sun Microsystems [4]. JMS is an abstraction of interfaces and classes that required by the messaging clients for communicating with messaging systems. Also, JMS products provide a solid base for creating enterprise messaging solutions and provide for secure transmission of critical information. It can be used for implementing e-commerce and enterprise application integration solutions. JMS combined with enabling technologies like XML can significantly increases capabilities of the enterprise messaging solutions, than when on their own.

The JMS application is composed of four prime entities which include *JMS provider*, *JMS clients* (both *producers* and *consumers*), *Messages* and *Administered objects*. The *JMS provider* is analogous to the event notification service in an event-based system (as discussed earlier) which is a message oriented middleware that implements the JMS interfaces. It is responsible for providing administrative and control features. *JMS clients* are analogous to the components (both producers and consumers combined) in an event-based system, and are the messaging clients in JMS. They are programs written in Java programming language, responsible for producing and consuming messages. *Messages* are the objects that communicate information between various JMS clients and finally, *administered objects* are preconfigured JMS objects that are created by an administrator for use by clients. The two kinds of JMS administered objects are *destinations* and *connection factories*. A *destination* is the object that a client uses to specify the target of messages it produces and the source of the messages it consumes. A *connection factory* on the other hand is the object a client uses to create a connection with a provider. Also, a JMS client that produces a message is called a *producer* while the JMS client that that receives a message is called a *consumer*. A JMS client can be both, a producer and a consumer as described in the previous section.

JMS supports two messaging models called *publish/subscribe* and *point-to-point* queuing models. These are also referred to as messaging domains. As the names suggest, 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 directly from one JMS client to another. The destinations in a publish/subscribe model are called *topics* and those in a point-to-point model are called *queues*. Elaborating on the publish/subscribe domain, a single producer can send a message to several consumers through a topic, which is a virtual channel to which consumer may subscribe to. Any message addressed to a topic is delivered to all consumers subscribed to the particular topic. Here, the producer sending the message is not dependent on the consumers receiving the message [10]. On the other hand, the point-to-point messaging model allows the JMS clients to send and receive messages both synchronously and asynchronously via the virtual channels called *queues*. Sender is the client that sends a message to the queue which is received by the receiver. The queue may have multiple senders but only one receiver at a time. The messages here are ordered as the queue delivers the messages in the order they were placed in it by the message server. As messages are consumed, they are removed from the head of the queue.

Messaging products are inherently asynchronous and hence there is no real timing difference between the production and consumption of a message. The JMS server acts as an intermediary container for storing messages that are sent from one client to another. However, messages can 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. This method blocks the receiving client until the message arrives or it times-out if the message doesn't arrive within the specified time limit. On the other hand, in asynchronous consumptions – the receiving client can register a *message 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 that acts on the contents of the message.

Apart from just being an event service, JMS covers a broad range of enterprise applications including enterprise application integration (EAI), push-models, etc. JMS also helps promote a robust, service based architecture by offering flexibility in integration, in the form of publish/subscribe and point-to-point functionalities.

2.3 Web Services

The functionality of interoperable machine to machine interaction over the web is primarily supported by a software system named Web Service [3]. These services comply with the W3C (World Wide Web Consortium) [11] and OASIS (Organization for Advancement of Structures Information Standards) standards [12]. The W3C develops open standards for interoperable technologies to lead the web to its full potential. The OASIS helps in the development, convergence and adoption of open standards for the global information society.

The Web Service software system is primarily composed of three participants namely: *service provider*, *service broker* and the *service requester*. The service provider is responsible for creating the web service and then advertising the service to all of its potential users by registering it with a service broker. The service broker maintains a registry (also called yellow pages) of advertised /published services and using this registry, the broker introduces the service providers to the service requesters. The service requester looks up the yellow pages for suitable service providers and then directly contacts the particular service provider to use its service [3]. The architecture of Web Services and the interoperation between these three participants is depicted in Figure 2.2.

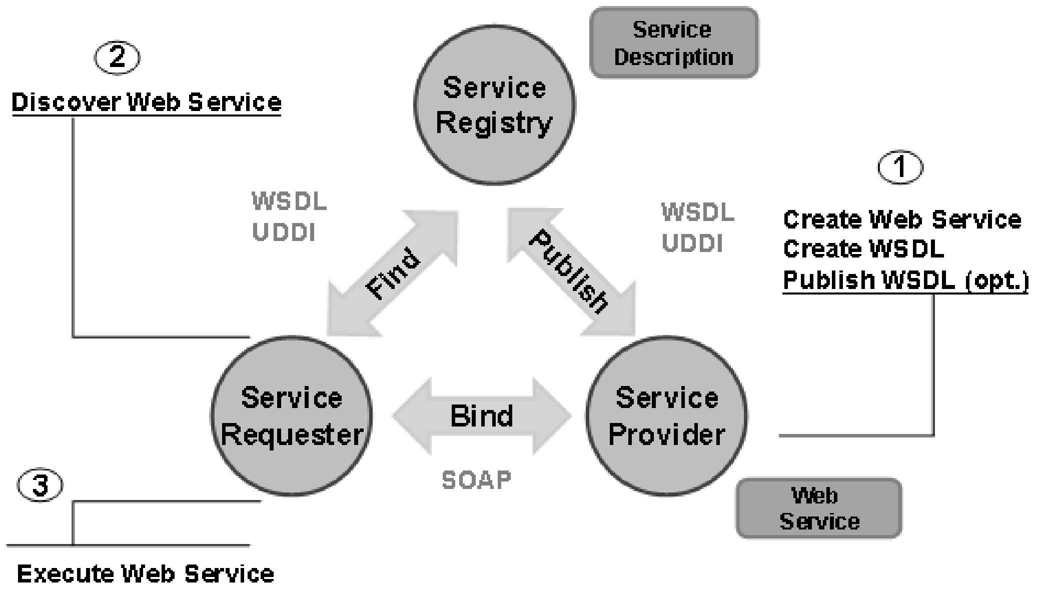

Figure 2.2. Web Services Architecture [13]

Web services has an interface described in a machine-process able format called the *Web Services Description Language (WSDL)* [3] which is and XML-based language used for describing the functionality offered by a web service. The WSDL provides a machine-readable description of how the service can be called, what parameters it expects and what data structures it returns. In simple words, it is analogous to the method signature in a programming language [14].

Simple Object Access Protocol (SOAP) [15] is an XML-based protocol specification for exchanging structured information in the implementation of web services in computer networks. A SOAP document is generally composed of three parts which include an *envelope,* a *schema* and a *convention*. An envelope provides a framework for a message describing what it is and how it is processed. A schema expresses instances of application-defined datatypes and finally, a convention represents remote procedure calls and responses. SOAP is implemented using the *Apache Axis (Apache eXtensible Interaction System)* which is an open source SOAP engine that provides a framework for producing web service producers and consumers. Axis is based on the standards "JSR (Java Specification Request) 101" and "Java™ APIs for XML based RPC" (also known as JAX-RPC) and runs as a servlet to process the incoming message, extract information from the message headers and to provide the RPC interface.

Universal Description, Discovery and Integration (UDDI) [16], is an OASIS yellow-pages standard that defines a way to publish and discover information about various web services as depicted in Figure 2.2. Whenever a particular service is registered with a UDDI, two SOAP messages are exchanged, first to establish authentication and secondly, to register the web service. The registry hence created is used as a service broker by various service requesters to identify and locate the services of their interest. UDDI is implemented using Apache’s jUDDI [17] which is an open source Java based implementation of the UDDI registry. jUDDI-enabled applications can look up services in the UDDI registry and then proceed to “call” the particular web services directly.

2.4 XML and JAXB

All the C2 (Command and Control) products (for example, OPORDs, OPLANS and FRAGOs, etc.) in this project are valid with respect to a common XML schema to make these XML documents understood by the Java technologies. JAXB is used to handle these XML documents.

2.4.1 Extensible Markup Language (XML). XML is a W3C standard that is used to provide for the sharing of structured data across the web [3]. It is classified as extensible because, apart from having its set of build-in elements, users can define their own elements (user defined elements). An XML must be well formed in the sense that the elements (e.g. tags, etc.) are properly nested and hence do not overlap each other. It is generally more useful for an XML document also to be valid with respect to a schema. Originally, an XML schema had to be expressed with a Document Type Definition (DTD) [3] which is a document that uses an extended form of BNF to specify the abstract syntax of conforming XML documents. Lately, most XML schemas are written in the XML Schema Definition Language which is in fact more powerful that the DTD language. For example, it allows the user to define new data types and handle namespaces naturally. Also, a document written in this language is in itself an XML document.

2.4.2 Java Architecture for XML Binding (JAXB). JAXB is a Java API that allows Java developers to map Java classes to XML representations [18].

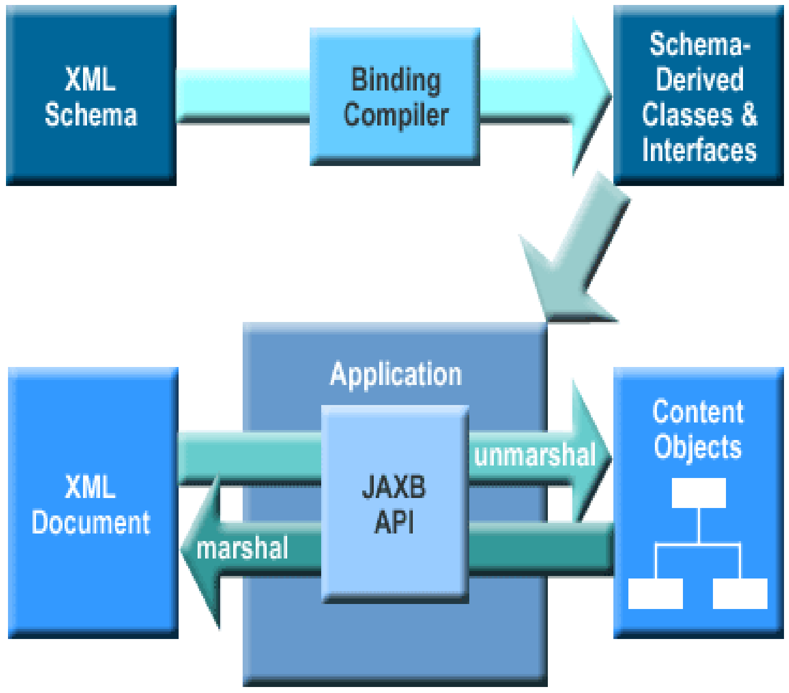

Figure 2.3. JAXB Architecture [19]

Figure 2.3 depicts the architecture of JAXB. JAXB generally provides two important features. It gives the ability to marshal Java objects to XML and the inverse, that is to unmarshal XML back into Java objects. In other words, JAXB allows storing and retrieving data in memory in any XML format, without the need to implement a specific set if XML loading and saving routines for the program's class structure [20].

2.5 Multiagent Systems

Multi-agent Systems (MASs) are systems composed of multiple interacting computing elements known as agents [21]. General consensus is that agents are computer systems capable of autonomous action that have the ability to interact with other agents [22]. Multi-agent systems are used in a wide range of applications and the field itself is interdisciplinary, influenced by various disciplines such as economics, philosophy, game theory, logic, ecology and social sciences. Since MASs derive their roots from so many diverse disciplines, there exist different views of multi-agent systems and the most common view is that they are basically systems of *interacting, intelligent agents*. Intelligent agents basically have three important characteristics in particular; they are *reactive*, *proactive* and have a *social ability* [21]. An agent is reactive if it can perceive its environment and respond to it rapidly. An agent is proactive if it exhibits a goal-driven behavior by taking the initiative, and, lastly, an agent is social if it is able to interact with other agents and humans. It is the interaction of various agents that creates a MAS. Also, each agent might have different roles, yet all agents collaborate to maintain or to achieve the goals of the system as a whole.

Interaction among agents allows multi-agent systems to exhibit complex behaviors and for these agents to collaboratively perform complex tasks, so they need to be able to communicate with each other [23]. Hence, there are several agent communication languages (ACLs) each of which is typically grounded in speech-act theory. Speech-act theory, analyze language use as consisting of acts such as asserting, requesting and commanding [2] explicitly represented by verbs such as “assert”,

”request” and “command”, respectively. These verbs are called *per formatives*. An older ACL for MAS is the Knowledge Query and Manipulation Language (KQML) which was developed as a part of the ARPA Knowledge Sharing Intuitive. KQML was intended for large scale knowledge bases that are sharable and reusable. Today, a much-used ACL is the FIPA-ACL is the proposed IEEE standard for agent communication [24] [25].

Foundation of Intelligent, Physical Agents (FIPA) is an IEEE Computer Society standards organization that promotes agent based technology and the interoperability of its standards with other technologies [26]. FIPA specifications represent a collection of standards that are intended to promote the interoperation of heterogeneous agents and the services that they can provide [26]. The FIPA Standards Committee currently has two kinds of contributing groups namely: Working Groups (WG) and the Study Groups (SG) [27]. FIPA has six working groups and these are established to carry out standardization projects within the scope of the FIPA SC [27]. One of the most important of the six working groups is the Agents and Web Services Interoperability Working Group (AWSI WG) which helps to fill the interaction gap between agents and web services.

The Java Agent Development Framework (JADE) [28] complies with the FIPA 2000 specification regarding communication, management and architecture. JADE provides the normative framework where FIPA agents can exist, operate and communicate while adopting a unique, proprietary internal architecture and implementation of key services and agents [2].

2.6 JADE

The Java Agent Development Framework (JADE) is a FIPA Standards compliant software framework that simplifies the development of multi-agent Systems. The JADE agent platform's basic middleware functionalities which are independent of the specific application, simplify the realization of distributed applications that exploit the software agent abstraction [2]. The framework is fully implemented in the Java programming language and consists of a runtime environment for agents, a library for Java classes that provide ready-made pieces of functionality, abstract interfaces for application-dependent tasks and a suite of graphical tools to help facilitate administration and monitoring of running agents [2].

The design objectives of JADE were primarily based on the agent abstraction [2] where, agents are autonomous and proactive and hence each agent needs its own thread of execution to control its life cycle, and to decide which action to perform when. Given that an agent has its own thread, the agent should not provide other agents with call backs to prevent other agents from co-opting control of their services. Message-based asynchronous communication is the most basic form of communication between agents since agents are loosely coupled and always have the ability to decline requests. Such a communication requires no temporal dependency between the sender and receiver, and allows receivers to decide which message to accept and which to decline. Also, a message is sent to an identified location, and the sender does not block waiting for the receiver to consume the message. An agent can join or leave the host platform at any time since the system here is peer-to-peer. Furthermore, each agent has a globally unique name and can be discovered by other agents through white-page and yellow-page services.

At the lowest level, a JADE platform consists of several agent containers that can be distributed over the network. Agents live in containers, which are the Java processes that provide JADE runtime support and all the services needed for hosting and executing agents. The main container, called “Main Container,” is the bootstrap point of a platform and is the first container that is launched. All other containers must join to the main container by registering with it [23].

2.7 JESS

Java Expert System Shell (JESS) is a rule engine and scripting language that supports both, rule-based and procedural programming [29]. Since it is dynamic and written in the Java programming language, it is an ideal tool for adding rule-based technology to Java-based software systems [29] such as JADE. A rule is structured as an *if-then* statement. JESS maintains a working memory where facts are stored. The *if* portion of the rule has conditions that have to be matched against facts in the working memory, and only then is the particular rule fired. The *then* portion of the rule specifies actions that may be performed when a rule fires. In a scenario where several if-conditions match, they are placed on the agenda and then, JESS uses a conflict resolution strategy to prioritize the rules and to determine which one fires.

The JESS rule engine uses the Rete Algorithm, which provides the functionality of matching the if-conditions (left-hand sides) of rules against facts in the working memory. The algorithm builds a network of nodes where, each node corresponds to a pattern occurring in a rule. The Rete Algorithm sacrifices memory for the rapid increase in speed.

It is a very common practice to use JESS to endow JADE agents with intelligence as both, JADE and JESS are Java applications. So, policies are encoded as JESS rules and JESS enforces these policies within JADE agents. However, despite all these advantages, there is one difficulty in using JESS with JADE agents. This pertains to the fact that 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. The solution here is to time-limit the execution of the rule engine.

2.8 JADE Gateway Agents for Web Services and JMS

Agent services are made available as web services with the help of a technology called *Web Services Integration Gateway (*WSIG*)* [22] and multiple agents are allowed to participate in the publish/subscribe model (as a Distributed Event-based System) with the help of another utility called *JMS Agent Gateway* [30]. This allows agents within a JADE platform to interact with a JMS provider.

2.8.1 Web Services Integration Gateway (WSIG). WSIG provides for transparent interconnectivity between the Web Services domain and the JADE platform. The WISG architecture includes the UDDI repository and JADE agents. WSIG helps expose the services provided by the JADE agents and publishes them in the JADE DF (the directory facilitator agent) as Web Services. This involves the generation of a suitable WSDL for each service-description registered with the DF [16].

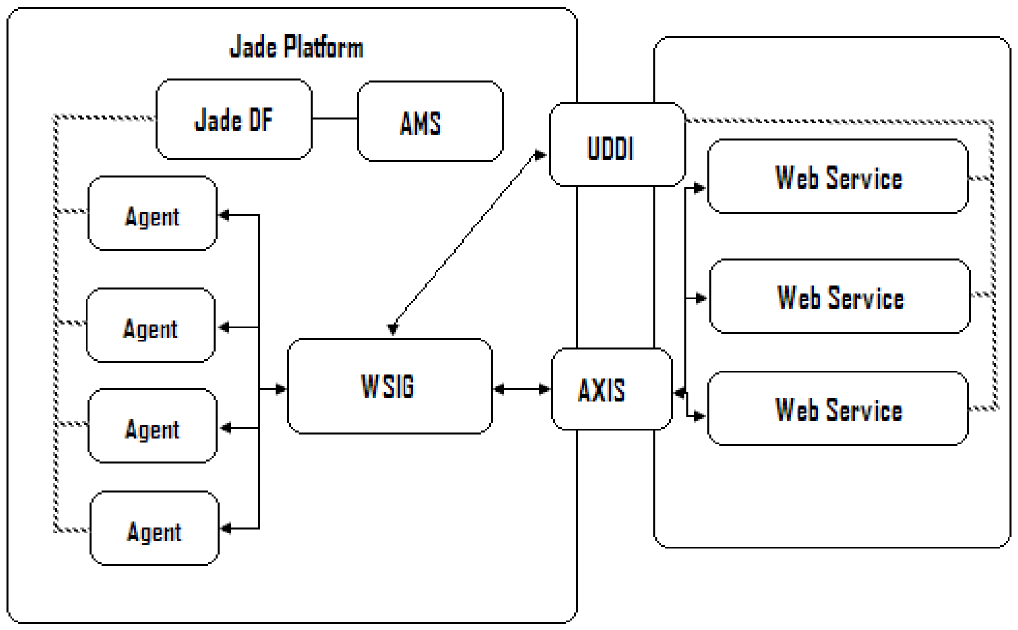

Figure 2.4. WSIG Architecture [31]

The WSIG architecture as shown in Figure 2.4 contains the standard JADE DF provided by JADE. Its functionality is modified to detect the registration (with an intention of publishing the Web service) of an agent service. These requests are first registered with the DF and then forwarded to the WSIG Gateway Agent which manages the entire WSIG system. There are two notable functions of the WSIG Agent Gateway. First, it receives the service registration from the JADE DF and then translates them into the corresponding WSDL descriptions, which is registered with UDDI yellow pages and second, it receives and agent service request from a web service client, finds the service in the UDDI repository, translates the request into ACL and then sends it to the requested agent. An agent response is translated into SOAP and sent to the requesting web Service.

2.8.2 JMS-Agent Gateway. This gateway introduces Enterprise Message Services (EMS) and their messaging models to agent-based systems. Its primary function is to provide for non-agent EMSs to be accessible to agents in an agent-oriented fashion. With direct access to the messaging models, an agent can easily interact with a variety of information dissemination services [23].

The JMS-agent Gateway primarily consists of a JMS *facilitator agent* that is responsible for receiving agent requests for the JMS provider and converts these requests into JMS actions. It is also responsible for making the connection to the JMS provider to perform subscription and message processing tasks on behalf of its clients. In summary, the primary role of the JMS facilitator agent is to publish messages to JMS destinations, to forward messages to consuming agents based in their subscription requests and to administer subscriptions [30]. Currently, the gateway supports 2 types of messages namely: `javax.jms.TextMessage` and `javax.jms.ObjectMessage.` Such message formats can carry FIPA ACL messages with different FIPA Content Languages (CLs) such as the Semantic Language (SL) and the Resource Description Framework (RDF). Since the gateway design uses the JMS API, any JMS-compatible EMS can interact with the gateway and provides the flexibility for the platform administrators to choose the most appropriate EMS implementation for their agent platform.

2.9 Orchestration and Workflows

The TITAN Work Flow Orchestration (WOS) service generates and executes workflows while interacting with other agents and TBSs (Titan Battle Command Support). Service orchestrations are analogous to service choreography. In service choreographies, the logic of the message-based interactions among the participants is specified from a local perspective whereas in service orchestration, the logic is specified from the global point of view of one single participant called an orchestrator [32]. Workflows are basically means to orchestrate services: Web Services can be orchestrated and multiagent systems can be choreographed. Here, WADE which is an extension of JADE, supports Workflows and the TITAN WOS service was implemented using it.

2.9.1 Orchestration. This is the means to structure concurrent activity since concurrency has become a practical problem with the growing use of multi-core, multi-cpu systems. It has become difficult for programmers to manage threads and locks explicitly. Also, these concurrency constructs do not serve well in expressing the high level structure of control flow in concurrent programs, especially while handling failure, time-outs and process terminations [33]. Hence, with so many issues in trying to achieve concurrency, many attempts have been made to rethink the prevailing concurrency models of threads with shared state and to put forth programming languages based on novel approaches. Some of these novel concurrent paradigms are discussed below:

Orc [34] is a novel language for distributed concurrent programming that provides uniform access to computational services including distributed communication and data manipulation. Using just four simple concurrency primitives, Orc helps a programmer orchestrate the innovation of sites to achieve the overall goal while managing timeouts, priorities, and failures.

Erlang is another concurrent programming language, designed by Ericsson [35] and primarily used for programming concurrent, real-time, distributed fault-tolerant systems. It is based on the Actor model [36], where every object is viewed as an Actor [37]. An actor contains a mailbox and a behavior, and this mailbox is buffered when two or more actors exchange messages. As an actor receives a message, its behavior is executed; the actor can send any number of messages to other actors and all of these communications happen asynchronously [35]. Hewitt, who developed the Actor model, points out [38] that hardware development is expanding in local and non-local concurrency, where local concurrency is used by new 64-bit multi-core processors and multi-chip modules whereas the non-local concurrency is used for wired and wireless broadband packet switched communications. Hewitt claims the all these developments favor the Actor model and the actors can reach the level of agents when they can express notions like those of intentions, plans, contract, belief and policy.

Oz [39] in another multi-paradigm language designed for advanced, concurrent, networked, soft real-time and reactive applications with support for any number of sequential threads dynamically created by the users. *Pict* [40] is yet another concurrent functional language and is similar to the Orc language. There are many more of similar concurrent programming languages in use today.

2.9.2 Workflows. A workflow is simply a process [41] which can be described as a collection of tasks, conditions and sub-processes. A process (sometimes called a procedure) can consist of a number of tasks that need to be carried out as a single resource. A resource is a generic name for a person, machine, or group of either persons or machines that can perform specific tasks. Each task is optional in the sense that it needs to be carried out only under certain circumstances, and the selection of which task needs to be executed depends on the order in which the tasks are performed. There are basically four constructs for routing: sequential routing, selective routing, parallel routing and iteration. With sequential routing, tasks are carried out one after the other. However, selective routing is when there is a choice between two or more tasks and the choice depends upon the specific properties of the case. Parallel routing is when several tasks can be carried out at the same time, i.e., in parallel. Finally, in iteration, a particular task is performed repeatedly until some condition holds.

A workflow is most commonly analyzed using: 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 – *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. A particular state is deemed reachable if there is a directed path in the reachability graph from the *start* state to it.

Van der Aalst defines a *soundness property* of a workflow net. This property is the minimum requirement that every process must meet. The following three conditions have to be satisfied for a workflow net to be sound. Firstly, for each token put in the place *start*, one (and only one) token appears in the place *end*. Secondly, when the token appears in the place *end,* all the other places are empty, and, finally, for each transition (task), it is possible to move from the initial state to a state in which that transition is enabled. A novel workflow language, *Yet another Workflow Language (YAWL),* was developed by Van Der Aalst and Ter Hofstede in 2002 [42], based on the analysis of the existing workflow management systems. YAWL is a Business Process Management

(BPM)/Workflow System that handles complex data, transformations and integrations with organizational resources. Petri nets [43] are used as a base, and mechanisms are added that allow more intuitive and direct support of the workflow patterns [41].

2.9.3 Workflow and Agent Development Environment (WADE). WADE adds to JADE functionality for execution of the tasks defined according to the workflow metaphor and a number of management mechanisms [44]. WADE is implemented through an eclipse plug-in called *WOLF,* which facilitates the creation of WADE-based applications by providing a graphical form for depicting processes in WADE. WADE adds to JADE the following two features: firstly, the capability to define agent tasks according to the workflow metaphor, and, secondly, additional components and mechanisms that facilitate the administration of a distributed WADE-based application in terms of configuration, activation and monitoring.

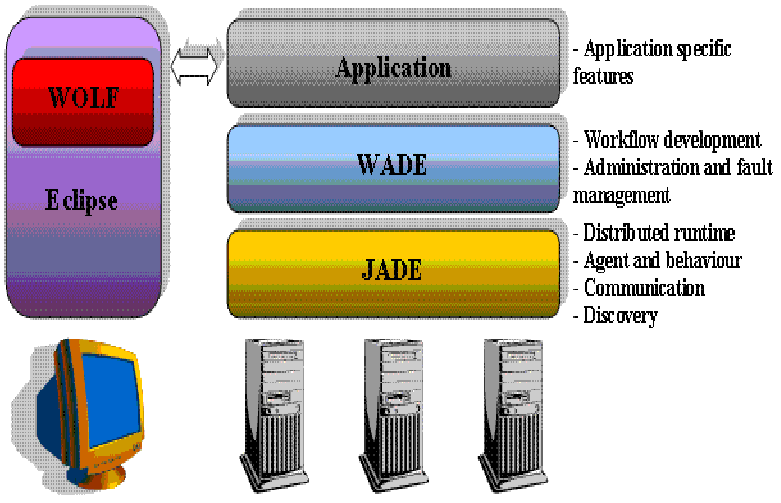

Figure 2.5. WADE Platform [44]

WADE's function is to bring the workflow approach from the business process level to the internal system logic level. WADE targets the implementation of the internal behavior of each single system rather than the high level services provided by different systems. Hence, WADE would be more suitable for addressing the execution of long and complex tasks, while the middleware technologies are more appropriate with systems that manipulate data in data repositories following user requests. The organization of the WADE platform is depicted in Figure 2.5.

WADE workflows are composed of activities like calling a Web Service,, invoking another workflow, and executing a raw code directly included in the workflow class. The various sets of activities in WADE can be classified into the following: Tool activities, Code activities, Web services activities, Subflow activities, Subflow join and Route activities.

2.10 Common Knowledge and Theory of Mind

Two of the prime concepts that support the coordination between TBSs are introduced here: common knowledge and theory of mind.

2.10.1 Common Knowledge. Common Knowledge is a prerequisite for coordination among artificial agents and humans. Generally, the initial common knowledge agents have is not sufficient for them to coordinate their actions in every possible situations there is. Hence, attaining common knowledge is very important [45]. Common knowledge is so central to coordination that it has been studied in various disciplines such as philosophy, linguistics, game theory and distributed systems.

To understand the concept of common knowledge, let us consider *G* to be a group of *n* agents, denoted by the ordinals *1, 2 … n*: *G* = {(*1, 2 … n*)} [34]. 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 $E_G^1\ \varphi \wedge E_G^2\ \varphi \wedge \ldots \wedge E_G^i\ \varphi \wedge \ldots$”. An intuitive example here would help us understand this better. Note that traffic lights would not work unless it were common knowledge that green means go, that red means stop, and that lights for opposite directions have different colors. If this were not so, we would not have the confidence to 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 a common knowledge. It can also be shown formally that coordination implies common knowledge [45].

Apart from characterizing common knowledge in terms of infinite conjunction, Barwise [46] identified two additional approaches namely: the fixed-point approach and the shared situation approach. In the fixed-point approach, we view $C_G\ \varphi$ as a fixed-point of the function [45] $f(x) = E_G(\varphi \wedge x)$. In the shared situation approach [47], given that A and B are rational, we may infer common knowledge among A and B thatφ, if the following three conditions are satisfied: 1. A and B know that some situation σ holds, 2. σ indicates to both A and B that both know that σ holds and 3. σ Indicates to both A and B thatφ. This approach is termed as the Mutual Belief Induction Scheme (Here we take mutual belief and common knowledge to be synonyms). Barwise concludes that the fixed-point approach is implied by the shared situation approach and is the correct analysis of common knowledge and that common knowledge generally arises via these shared situations. H. H. Clark and Carlson [47], 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 Clark [48] 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.

2.10.2 Theory of Mind. This is the ability to attribute mental states (beliefs, desires, intentions, assumed roles, etc.) to one another and to understand that others’ have mental states that are different from one’s own. Early work on Theory of Mind (ToM) showed striking improvement in false-belief (FB) and Level 2 visual perspective-taking (PT) aspects among children between ages three and five. An example of an FB task is where a child watches as a puppet see a cookie put in one of the two boxes and leave, someone moves the cookie to the other box, and when the puppet returns, the older child (having the notion of false belief), but not the younger, says that the puppet will look in the original box. An example of a PT task where the older child but not the younger one, understands that a picture book oriented correctly for them on the table looks upside down for a person sitting on the opposite end of the table. Theory of Mind 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.

Regarding ToM and TITAN, in the intended use of TITAN, we know that each unit has its own copy of the TITAN suite of services called TBS (Titan Battle Command Support). Since there is a hierarchy of commanders, there needs to be a hierarchically integrated set of TBS's where the parent in the hierarchy coordinates with its children. Coordination of child TBSs with a parent TBS using the same paradigm as used within a TBS – a multi-agent system is not feasible because of the communication requirements. So inter-TBS communications use JMS. It is already proposed [19], that the appropriate TITAN services in a TBS maintain a WADE workflow involving agents that are proxies for its child TBS. This workflow determines what JMS messages are published for the child TBSs, and the JMS messages published by child TBSs result in updates to the proxy agents maintained by the parent TBS. It is claimed here that, use of proxy agents in this way implements a scaled-back version of ToM in developmental psychology [49]. Also, ToM (it is claimed) provides an essential means for a group to achieve CSK [50].

2.11 Holons and Holarchies

The bare notion of MAS includes nothing about the structure of the system into groups at various levels. To capture such notions, we introduce the concept of a *Holon*. As Arthur Koestler formulated the term, a *holon* is an identifiable part of a system that has a unique identity yet is made up of sub-ordinate parts and in turn is part of a larger whole [51]. Koestler used the term "holon" to name recursive and self-similar structures in biological and social entities. In a society, each individual male is a holon and each female is a holon too. The relationship between a male and a female is also a holon, yet each of these bipolar units is part of the family holon, and so on. Entire organs such as the kidney and heart are capable of continuing their functions to some extent as independent wholes, when isolated from the organism and supplied with proper nutrients.

A holon can also be seen as a node in a tree structure. The main characteristics of a holon is that it asserts its individuality in order to maintain the set order in the tree structure, but it also submits to the demands of the whole tree structure (the system) in order to make the system feasible. Holons are self-contained, autonomous pieces. A holon is both a whole (self-assertive and self-reliant) and a part (cooperative and integrative). This duality is described as the "Janus effect" and is similar to the particle/wave duality of light [51]. The Janus-faced entity shown in Figure 2.6 illustrates a holon looking inward, seeing itself as a self-contained unique whole, and looking outward, considering itself as a dependent part. Its self-assertiveness is the dynamic sign of its unique wholeness, its autonomy and its independence as a holon. Its integrative tendency, in contrast, expresses its dependence on the larger whole to which it belongs and is just as important as its self-assertiveness.

A holon cooperates and integrates with the higher level holon. Holons can also receive instructions or submit to higher level holons. A holon coordinates or sometimes competes with the holons at the same level, and directs the lower holon immediately below it. The self-reliant characteristic ensures that holons are stable and able to survive disturbances. The subordination to higher level holons ensures the effective operation and cooperation of the larger whole. A holon at any level is not necessarily the sum of its subordinates. It is a part of something bigger that it is being influenced by. It has a high degree of autonomy and, to some extent, an independent life of its own.

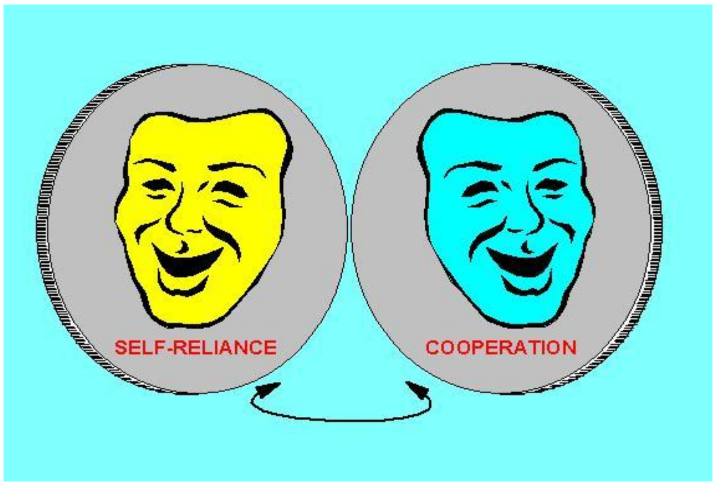

Figure 2.6. A description of the "Janus Effect"

The characteristics of holons at one level do not represent the characteristics of the level above or below them. The further down the tree (holarchy), the more mechanized, stereotyped, and predictable the behavior. Higher level holons have more flexibility and function in a more abstract state—see Figure 2.7.

Holons and agents are comparable. An agent is an entity that is autonomous, social, reactive, pro-active, rational, mobile, and capable of learning [52]. A Holon is an element that has the following behavioral properties: autonomous, cooperative, and recognizable. Holons, however, also possess structural properties such as being recursive (holons are made up internally of self-similar holons, and so on until an atomic level is reached) and being dynamic. In contrast, an agent is an autonomous entity that is not composed of other agents. In fact, we can view an agent as a leaf of a hierarchy of holons (or holarchy - see Figure 2.8).

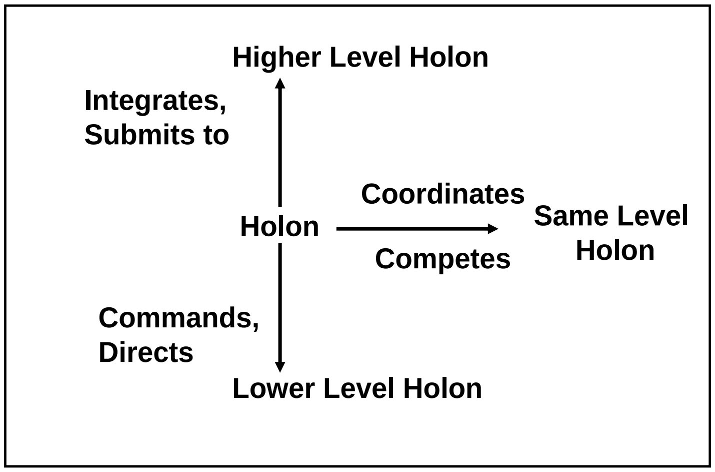

Figure 2.7. Holon relations

Holonic systems (HSs) and multiagent systems have been studied by different research communities. HS research is inspired by flexible manufacturing tasks while MAS research is motivated by programming of distributed systems. HS is a manufacturing-specific approach for distributed intelligent systems and is oriented toward low-level behavior and communication standards while MAS is a broad software approach used for distributed intelligent control and is oriented toward behavioral models, cooperation and coordination strategies, and learning [53].

"Holarchy" was coined by Arthur Koestler as a combination of the Greek word "holos," meaning whole, and the word "hierarchy". It is a hierarchically organized structure of holons. A holon appears as a whole to the parts under it in the hierarchy, but it appears as a part to the wholes above it. Therefore, a holarchy is a result of a whole that is also structured of parts that are themselves wholes.

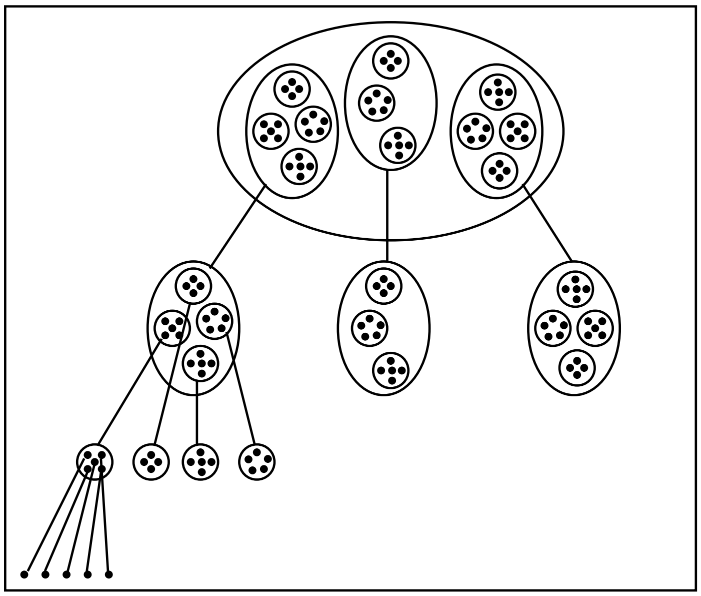

Figure 2.8. Holon characteristics [53]

The strengths of a holarchy include its ability to construct very complex systems that use the resources efficiently, its high resilience to both internal and external disturbances, and its adaptability to environmental changes. All these characteristics can be clearly observed in biological and social systems.

The stability of holons and holarchies stems from their self-reliant quality and their independence in handling situations and problems at their particular level without assistance from higher-level holons. Thus, for example, a single cell modifies its structure to gain its most favorable relationship with its environment.

To fix a conflict between holons, step up to the next higher whole and establish more integration and cooperation among its parts. For example, a conflict between two people cannot be resolved by just looking into their individual minds, but step up and examine what kind of relationship they have, or what kind of group they are a part of, then work to establish cooperation. It is possible to optimize a certain whole by re-aligning its parts. Looking at one holarchy or one holon at a time can be valuable to decipher the model.

2.12 Parts-whole state charts and State Flow

2.12.1 Statecharts. Statecharts were developed by David Harel in the 1980s. This formalism was originally created to support the Israeli aircraft industry in building airplane simulators [54]. It is an extension of deterministic finite state automata (or finite state machines, FSMs), which is an important topic in computer science. Statecharts are used for managing complex designs and for graphically simulating complex reactive systems. Statecharts show the state of one or more entities, how they transition between states, and how they interact with each other. Statechart diagrams contain states, transitions, events, and actions. Statechart diagrams capture the system's dynamic behavior (event-driven), model reactive objects (user interfaces, etc.) and entities' lifecycles [55]. The state diagram is very important for the developer in order to understand the association and hierarchy of entities.

Statecharts are a visual formalism used for describing the behavioral of complex, concurrent systems. Statecharts are also utilized by members of system development teams as a powerful communication tool to handle the complex relationships of subsystems within a larger system. Statecharts also reflect important OO issues such as inheritance.

A Statechart is a complete graphical description of the system's performance [55]. It consists of discrete states and transitions. Statecharts describe how entities communicate and collaborate while carrying out their own internal behavior. Statecharts extend classical state diagrams by allowing the representation of hierarchical and concurrent active states. Statecharts also enable clustering, concurrency, refinement, and "zooming" capabilities for ease of navigation between levels of abstractions. Statecharts extend finite state machines to support the following basic principles: hierarchy, broadcasting, and history [55][56]. Concurrency, broadcasting and history are key properties of a hierarchical Statechart and could be used at the same time.

Statecharts describe the system's dynamic behavior over time by modeling its phases. The main elements used in Statechart diagrams are states, transitions, events, actions and activities. States and transitions define all possible states a system can reach during its lifetime. The system reacts to events by changing its state. Recently, Statecharts have been adopted into the Unified Modeling Language (UML) as one of its standard methods. Several extensions to the basic Statechart notation, such as *Parts/whole Statecharts* (see section 2.12.2), have been proposed.

2.12.2 Parts/Whole Statechart. The Parts/whole Statechart notation is a conservative extension of Statecharts introduced by Lucca Pazzi [57][58]. Parts/whole Statecharts are used for the explicit representation of compound behaviors of reactive systems. Parts/whole Statecharts use an explicit approach to modeling aggregate entities by introducing a coordinating whole as a concurrent component on a par with the coordinated parts. Parts/whole Statecharts introduce no new notation but simply put constraints on how concurrent components coordinate and require a specific section for the whole and different sections for the parts.

Statechart formalism is characterized not only by hierarchy (XOR states) and orthogonally (AND states) but also by broadcast communication: An action in one orthogonal component can serve as a triggering event in other components. This is one way components of a Statechart are implicitly coordinated. Other ways are by transition conditions that test the state of another component and by using the same event as a trigger in transitions in different orthogonal components [57]. This implicit coordination introduces a web of references and eliminates encapsulation, adversely impacting the reusability, understandability, and extendibility of the resulting abstractions. We therefore follow Pazzi [57] in using an explicit approach to modeling aggregate entities, introducing a *whole* as an orthogonal component on a par with the parts that are coordinated. A part communicates only with the whole, never with other parts. The resulting *Parts/whole Statecharts* introduce no new notation but simply put constraints on how orthogonal components coordinate and require a specific section for the whole and different sections for the parts. Statecharts usually do not involve the decomposition of aggregate entities to more than one level, but with Parts/whole Statecharts this becomes natural. A Parts/whole Statechart may thus support two kinds of hierarchies, XOR hierarchies and parts/whole hierarchies, the latter modeling holarchies.

2.12.3 Stateflow. Stateflow is an interactive graphical design and simulation tool for event-driven systems. Stateflow provides the language elements required to describe complex logic in a natural and comprehensive form. It is integrated with MATLAB, Simulink and optionally with the Real-Time Workshop (RTW) – see Figure 2.9. It provides an efficient environment for designing complex reactive systems and enables the graphical representation of hierarchical and parallel states as well as the event-driven transitions between them. Stateflow augments traditional Statecharts with many innovative capabilities and can automatically generate C code using its Coder [59].

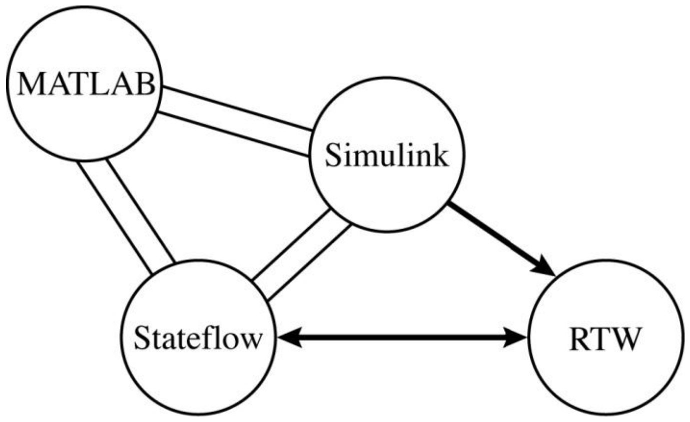

Figure 2.9. Relation between Stateflow, MATLAB, Simulink, and RTW

Our primary use of Stateflow is to animate the Statecharts representing the states and the data of our reactive systems for enhanced understandability and easier debugging. Other key features of Stateflow are that it provides language elements, hierarchy, parallelism, and deterministic execution semantics for describing complex logic [59]. Stateflow also defines functions graphically, using flow diagrams; procedurally, using the MATLAB language; and in tabular form, with truth tables.

CHAPTER 3

TITAN PROGRAM

The Tactical Information Technologies for Assure Net Operations (TITAN) Program is an Advanced Technology Objective (ATO) funded by the Army and described in *TITAN ID&M: A Guide for Developing Agent-Based Services* [1] – currently the definitive document on the TITAN Program. The goal of this ATO is to demonstrate how emerging information technologies can significantly improve key areas of tactical operations, ultimately resulting in the transition of software developed under the ATO to existing battlefield systems. The key areas of tactical operations, targeted for improvement are: Information Dissemination and Management (ID&M) Network Management (NM) and Information Assurance (IA).

The TITAN ID&M sub-project primarily 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 that is compatible with the Joint Consultation, Command and Control Information Exchange Data Model (JC3IEDM) [60]. This aims to enable the interoperability of systems and projects that share command-and-control [61]. The NM sub-project addresses the problem of network reliability and bandwidth capacity which are very important for timely dissemination of data. Lastly, the IA sub-project addresses the problem of security and sensitivity of tactical information that is disseminated.

Elaborating further on the TITAN ID&M, the Battle Command Services mentioned here satisfy the need to create *operational orders (OPORDs)* as the output from mission planning activities as well as to update them by producing *fragmentary orders (FRAGOs)* during mission execution. These OPORDs from commanders are used to synchronize military operations where each OPORD contains descriptions of the task organization, the overall mission, administration and logistic support, the situation, mission execution and finally the commands and signal for the specified operation. OPORDs are an important kind of command and control (C2) products [62], which are directives, issued by a commander to his/her subordinate commanders in order to coordinate the execution of an operation. Yet another important C2 information type is the *operational plan (OPLAN)* which is a proposal for executing a command to conduct a military operation. Commanders may initiate preparation of possible operation by first issuing an OPLAN. FRAGOs and *warning orders (WARNOs)* are two more types of C2 products. FRAGOs provide timely changes of existing orders to the subordinates and notifications to higher commanders. On the other hand, WARNO is simply a notice of an order/action that is to be followed. An XML schema is used in the TITAN Program to represent the above mentioned C2 products and the root XML element of each C2 product document is `prdC2`, whose type attribute specifies the type of C2 product and whose `dsg` attribute specifies the designator (authoritative source) of the C2 product.

Figure 3.1 shows a hierarchically integrated set of the C2 systems. The Information Management component of each of these systems support the capability to disseminate information between C2 systems, operating in different echelons (levels of command) as shown by the green arrow in the Figure 3.1. There is exactly one active instance of an OPORD as well as a collection of other related artifacts like FRAGOs and WARNOs, associated with each instance of a C2 system.

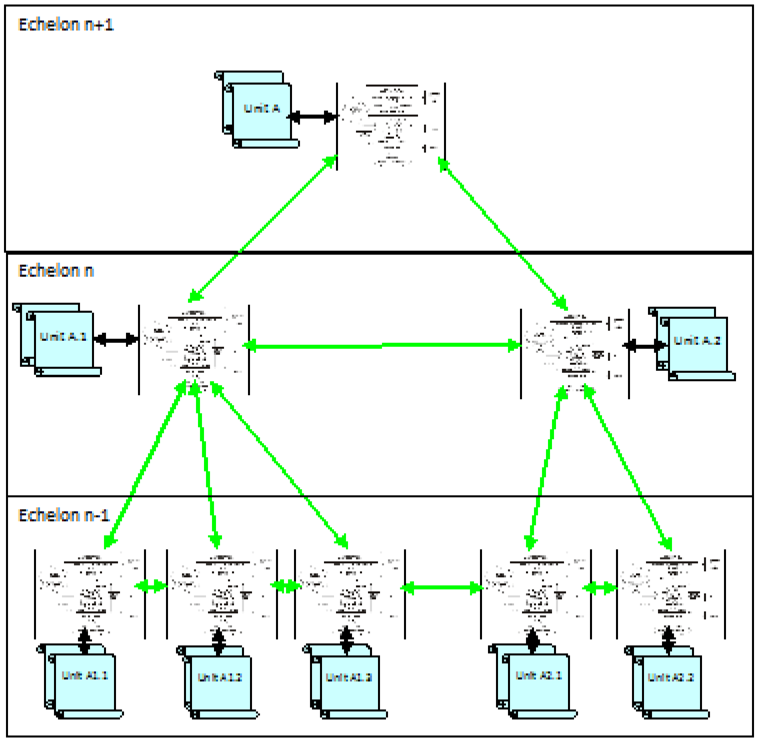

Figure 3.1. Information Dissemination [1]

The major activities performed by the C2 system are planning, preparation, execution and assessment. Planning handles the task of assembling information into an OPORD and then distributing the OPORD to any party whose behavior could be influenced by that OPORD. Preparation involves exchanging information to cross-check the current state of forces against the state assumed in the OPORD. During execution, ID&M involves exchanges of information needed to keep track of mission progress and finally, assessment is an activity that supports the other three activities, for example, by monitoring the planning, preparation and execution phases.

A TITAN Service is generally referred to as a *TITAN Battle Command Support (TBS)* It is a special type of *federate* defined to be an executing program that is aware of other federates and is able to interact with them via a common communication infrastructure, which in case of TITAN is the Java Messaging Service (JMS) [4]. Since there is a hierarchy of commanders, there needs to be a hierarchically integrated set of TITAN suites where each TBS provides four different types of interfaces, namely 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) [1] and the Battle Command Standards for security, when publishing information for consumption outside of the tactical operation center (TOC). As far as JMS topics go, the TBS uses several JMS interfaces, one of which is the JMS interface to other TBSs and to Battle Command War-Fighter (BCW) - a user interface that either displays information to, or receives manual information from a human consumer with TBS capabilities.

Figure 3.2 depicts the services planned for each TITAN suite under the ID&M part of the TITAN Program, three out of which are implemented here. The three implemented services are:

- the Alert and Warning System (AWS) [63], which generates warnings, alerts and predictions,
- the Workflow Orchestration Support (WOS) [64] Service, which organizes the workflow of services across the enterprise for synchronization and

- the OPORD Support Service (OPS) [65] which provides support services to manage C2 products like OPORDs and OPLANs

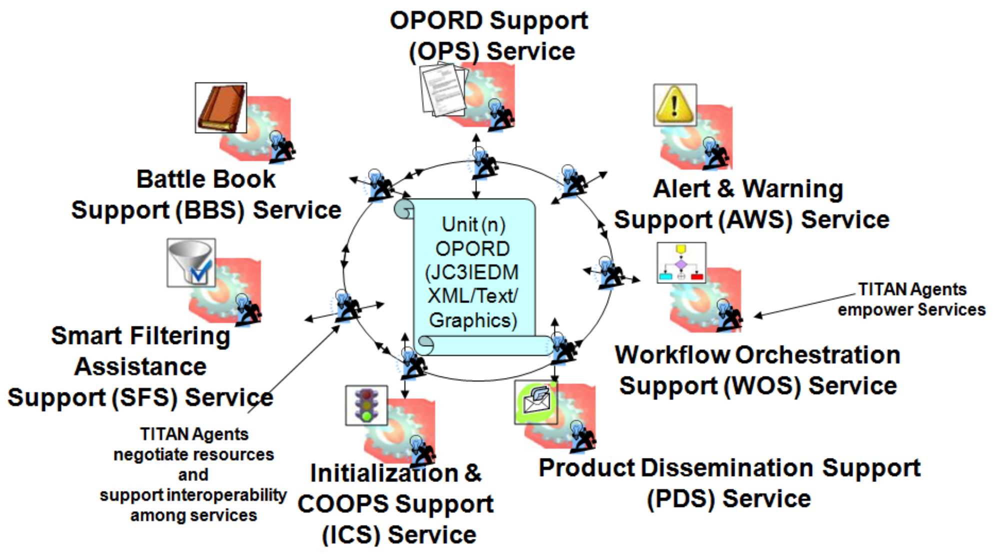
OPORD Support (OPS) Service
Alert & Warning Support (AWS) Service
Battle Book Support (BBS) Service
Unit (n) OPORD (JC3IEDM XML/Text/ Graphics)
Workflow Orchestration Support (WOS) Service
TITAN Agents empower Services
Smart Filtering Assistance Support (SFS) Service
TITAN Agents negotiate resources and support interoperability among services
Initialization & COOPS Support (ICS) Service
Product Dissemination Support (PDS) Service

Figure 3.2. TITAN Services [1]

Each TITAN service session implements the state model shown in Figure 3.3. The state transitions show the relationships between the TITAN service session states and the C2 system activities. Each TITAN service session implements the state model shown in Figure 3.3. 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 then communicates with other systems in preparation for supporting C2 activities. Assessment is performed in all states except the initialize state.

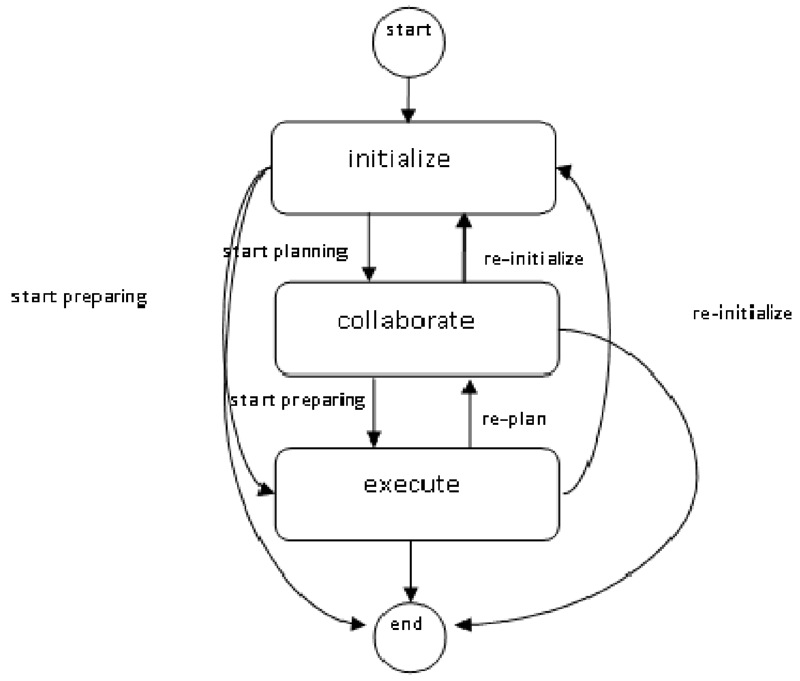

Figure 3.3. State Model of a TITAN Service Session

The four JMS topics to which all messages are published are *TITANcontrol* for initialization, *TITANplan* for collaboration, *TITANexecute* for tasking and situational awareness and *TITANassess* for alerts and warnings relevant to Commander’s Critical Information Requirements (CCIRs).

The three implemented TITAN Services are elaborated in the following sections of this chapter.

3.1 The AWS Service

AWS is implemented as a single Dispatcher (Coordination) Agent that provides for coordination and monitoring of roles of a suite of assessor agents, each handling a single instance of a Commander's Critical Information Requirements (CCIR) of a specific type. The three types of assessor agents are [63]: The *Area Protection Assessor* agent that evaluates potential threats to a polygonal area defined in the CCIR, the *Route Protection Assessor* agent that evaluates potential threats to a route defined in the CCIR as a sequence of line segments and the *Event Cluster Assessor* agent that identifies clusters of Improved Explosive Devices (IEDs) and generates an alert report of appropriate severity if the number of IEDs in any cluster exceeds a CCIR-defined threshold.

The Dispatcher Agent that is implemented here provides alerts and warning through a user interface, as it monitors a very simple simulation of the execution of an OPORD. The user interface indicates whether an alert report should be generated by the Dispatcher Agent through this interface. Also, the Dispatcher Agent communicates by receiving ACL messages from an assessor agent to determine the alert level.

3.2 The OPS Service

According to the guide, the OPS service is to be achieved with the help of two Cougaar agents, which are [65], the *Message Agent* for handling the JMS messaging in the system and the *Update Agent* for performing all operations on the OPORDs and OPLANs. The *Guide's* description of these agents requires *plugins* and *blackboards* where, plugins are analogous to JADE agents and blackboards are shared memory that are used for plugin communication within and between agents.

The Message Agent contains the JMS plugins and takes JMS messages of the various JMS topics and inserts them on the blackboard. These messages are encoded with the CERDEC's `prdC2` schema. On startup, this agent receives OPLANs and OPORDs via the TITAN control JMS topic and during collaboration and execution - it receives reports, WARNOs, FRAGOs, OPLANs, and OPORDs. The Message agent relays from its blackboard, the 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 a *Versioning* plugin, which subscribes to the blackboard for all messages that the Message Agent's JMS plugin, publishes to the blackboard. The Versioning plugin republishes each of these messages and depending on the message type, various other plugins will operate on them. It also replaces the 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 some other plugins such as the *Assess* plugin and the, *Plan* plugin.

3.3 The WOS Service

The WOS generates and executes workflows from tasks and other trigger objects and archives messages to support status requests and situational reasoning [64]. There are mainly two gateway agents that support communication outside the WOS service: the SOAP Gateway Agent which provides a SOAP service accessible to 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.

The WOS, apart from the two gateway agents mentioned above, is composed of three collaborative agents namely the Orchestration Agent, the Workflow Agent and the Monitor Agent. The Orchestration Agent receives tasks and other trigger objects and generates workflows according to the business roles for those objects. The Workflow Agent executes the workflows while interacting with other agents and TBS services, and, finally, the Monitor Agent receives all incoming messages from the two gateway agents and routes outgoing messages to the respective gateway agents.

CHAPTER 4

ARCHITECTURE AND IMPLEMENTATION

This section discusses the implementation of the Assessor Agent which is a part of the AWS service presented in the previous chapter. The implementation here builds on the work done by two previous MS students who worked on this project: Kiran Krishnamurthy [23] and Srinivas Reddy Banda [19]. The initial implementation of the AWS and the OPS and their intercommunications was carried out by Krishnamurthy. The AWS as we know from the previous section consists of a single coordination agent called the Dispatcher Agent and a suite of assessor agents, each handling a single instance of the OPORD. The three important assessor agents defined under the TITAN project are: the Area Protection assessor agent, the Route Protection assessor agent and, the Event Cluster assessor agent. Krishnamurthy implemented a very basic yet functional stub of the AWS, where an assessor agent was simulated in a very general sense as a user interface, with a provision to input the alert or warning as text that would be raised and propagated as an alert through the system, by the Dispatcher Agent.

The work implemented here extends the functionality of the Assessor Agent to help identify the potential threats (e.g. IEDs) in the particular phase of a mission and hence, help provide the shortest and safest path through the area of operation during a particular phase of the TITAN mission. Here, Dijkstra’s shortest path algorithm and the convex hull algorithms are implemented as the main computational blocks of the Assessor Agent. Dijkstra’s algorithm provides the shortest path through the area of operation of the phase, given all the possible way points from the start to the end of the area of operation. The algorithm implemented here operates dynamically so as to emulate the dynamic environment of a battlefield scenario. So, as the commander learns about new way points or potential threats (e.g. IEDs) during a phase, the algorithm reiterates and computes a new shortest safe path to the destination of that phase. The convex hull algorithm here helps create a hull around a region of high IED density. A path along the circumference of the hull can be considered safe for a commander to take, during his/her pursuit of reaching the destination in the area of operation of the phase. If at any point during a phase a commander detects a group of IEDs along the initial shortest path generated by the algorithm, a convex hull is formed around the IEDs, and the shortest path along the circumference of the hull is computed and then, a new shortest path from the end of the hull path to the destination of the phase is determined.

The implementation also maps both the shortest path through the way points and the convex hull formed around the IEDs onto an applet, which provides a graphical representation of the most efficient (shortest safe) path. This serves as a virtual map for a commander using it to navigate through the area of operation of a particular phase.

Banda , apart from implementing the WOS service and its intercommunication with the AWS and OPS services, simulated the TBS operations in a distributed environment, i.e. implemented all of the TBS services mentioned above on multiple (in this case, three) systems. This setup provided for the simulation of multiple phases of a mission, aiding the commanders who are in charge of the phases to navigate through their respective phases of the mission.

The work implemented here brings in the aspect of synchronization into the communications between the TBSs on multiple systems. The above mentioned algorithm along with generating the shortest safe path through a phase of the mission, estimates the time it would take for the commander/troop to get through the phase i.e. the time it would take to navigate from the start-end of the area of operation of the phase, provided the average speed of the troop is known. This timing information is vital for synchronizing operations of the troops in multiple phases of the mission.

This implementation was carried out in the Java language. The version of JADE used requires versions 1.6 of the JDK (Java Development Kit) and the JRE (Java Runtime Environment) to be installed on the workstations. The applications required to run the current implementation of the TITAN services include the JBoss Application Server version 5.1.0.GA (to furnish both the JMS Provider and the web server) and JADE version 3.6 (to provide the agent frame work). For the two gateways to allow interoperation among the three technologies we use the JMS-Agent Gateway add-on version 0.5 and the WSIG add-on version 2.0. The latter requires a UDDI repository; for this, jUDDI version 0.9rc4 with MySQL version 5.1.34 as the relational database management system was used. WADE version 3.0 is used for executing workflows. Finally, Java AWT and applet APIs are used to present the output of the assessor agent graphically.

The following section discusses the architectural background of the TITAN project that is needed to be setup and running before the Assessor Agent of the AWS service can be implemented. The second section in this chapter shows how the Assessor

Agent was implemented and made to coordinate with the TBSs on multiple systems. The final section of this chapter talks about the challenges encountered during setup and implementation.

4.1 Architectural Background

4.1.1 Overview of TITAN Services. The three TITAN services (AWS, OPS, and WOS) developed for this project involved integrating three different paradigms of concurrency and information dissemination: multiagent systems, Web Services and event-based systems. These paradigms are integrated and effectively used for communication among the agents providing these services.

The JADE agent framework was used to implement the prototypes of some of the agents envisioned for the OPS, AWS and the WOS services in the TITAN program. Two main JADE agents, the Message Agent and the Update Agent, provide the OPS service and a single Dispatcher Agent along with potentially a suite of assessor agents (but just a single one in the current implementation) provides the functionality of the TITAN AWS service. Also, the TITAN WOS service functionality is implemented with the help of three more JADE agents: the Monitor Agent, the Workflow Agent and the Orchestration Agent. In addition, there is the JMS Gateway agent and the WSIG agent, that interact with all of the above mentioned service agents. Recall that the JADE agents work only on a single instance of an OPORD at a time to provide their services.

Recall that the JMS API defines a common set of interfaces and associated semantics that allow programs written in the Java programming language to communicate with other messaging implementations. Using the JMS-Agent Gateway, JADE agents are able to use the JMS APIs in order to create, send, receive and read messages, allowing the agents to use the publish/subscribe messaging model for communication across TBSs. The service agents mentioned in the previous section, use this model for communication by publishing messages and subscribing to various JMS topics. The publish/subscribe model provides an event-based approach for communication and loose coupling among the communicating entities, both of which are desired in the larger TITAN system, which generally involves multiple child TBSs for each parent TBS in the hierarchy.

In the OPS service as implemented by Krishnamurthy, the Message Agent is responsible for handling the JMS messaging. The Message Agent subscribes to three JMS topics, TITANcontrol, TITANaccess and TITANblackboard and is a publisher to two tipics, TITANexecute and TITANblackboard. The Update Agent subscribes to the 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 OPLAN/OPORD from the TITANblackboard which is again published to TITANexecute so that the agents are aware of all the updates made the others. 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 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.

The WOS service implemented by Banda consists of one JADE agent called the Monitor Agent, and 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. 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 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.

The current implementation of the AWS service by Krishnamurthy 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 an input to simulate the occurrence of an alert situation and makes it available to the Dispatcher and 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.

The work reported here extends the implementation of the Assessor Agent by replacing the execution of the OPLAN/OPORD that is simulated through a user interface, by a fully functional Assessor Agent that takes the possible way points listed in the particular instance of the OPLAN/OPORD and computes the shortest path (using Dijkstra’s shortest path algorithm) to the destination of the area of operation of a particular phase. If during the phase, the commander learns about the existence of a group of IEDs on the shortest path to the destination, the Assessor Agent forms a convex hull around this IED group and estimates the shortest path around the convex hull, ultimately generating a new path from the end of the path around the convex hull to the destination of the area of operation of the phase. All this dynamic route learning and computation is communicated to the commander in the form of a map plotted on a Java applet as shows in the penultimate section of this chapter.

4.1.2 Publish/Subscribe Messages using the JMS Agent Gateway. The JMS Agent Gateway allows JADE agents to communicate with other JADE agents located in other 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 JMS topics, and receive messages based on their subscription requests. This 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 messages onto a JMS topic.

The ontology defines the vocabulary and the semantics used while sending and receiving a JMS message [66]. 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. 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. To subscribe, one first creates 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. One then sets the *`Subscription`*, specifying the destination and the subscription's durability and adds a new `JmsSubscriptionInitiator`, passing in the above subscription message.

4.1.3 Agent Services as Web Services. Recall that WSIG [52] 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-Descriptions, 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.

4.1.4 Integration of WOS, AWS and OPS Services. The integration of the three services is achieved 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 is with the other TBSs but also includes reports from the AWS in the current TBS and updates to the OPORD from the OPS also in the current TBS. The original OPORD is provided directly to the Orchestration Agent. Communication that would go to the parent is simply displayed or stored on file. Updates that would come from the parent are supplied as user input. This approach provides a clear partition of the system. A history of the unit's message activity is 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 can 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. Finally, it is the intercommunication between the several TBSs.

The implementation of the communication between many of these above mentioned TBSs is also important for the proposed work to be useful in a multi-hierarchical army structure consisting of a group of commanders each expected to co-ordinate with the other in the most efficient way possible. This communication between commanders is analyzed and simulated as state flow in chapter 5.

4.2 Implementation of the AWS Assessor Agent

This section discusses the implementation of the Assessor Agent that was described in the beginning of this chapter. As the initial input, the Assessor Agent reads the co-ordinates of the way points through the area of operation of a particular phase from a single instance of the OPORD that is received by the TBS at that particular instant. The Assessor Agent takes these way points and determines the shortest path from the source to destination of the area of operation of the particular phase with the help of the Dijkstra's shortest path algorithm. The computed shortest path is mapped onto a graphical interface through a Java applet as shown in Figure 4.1.

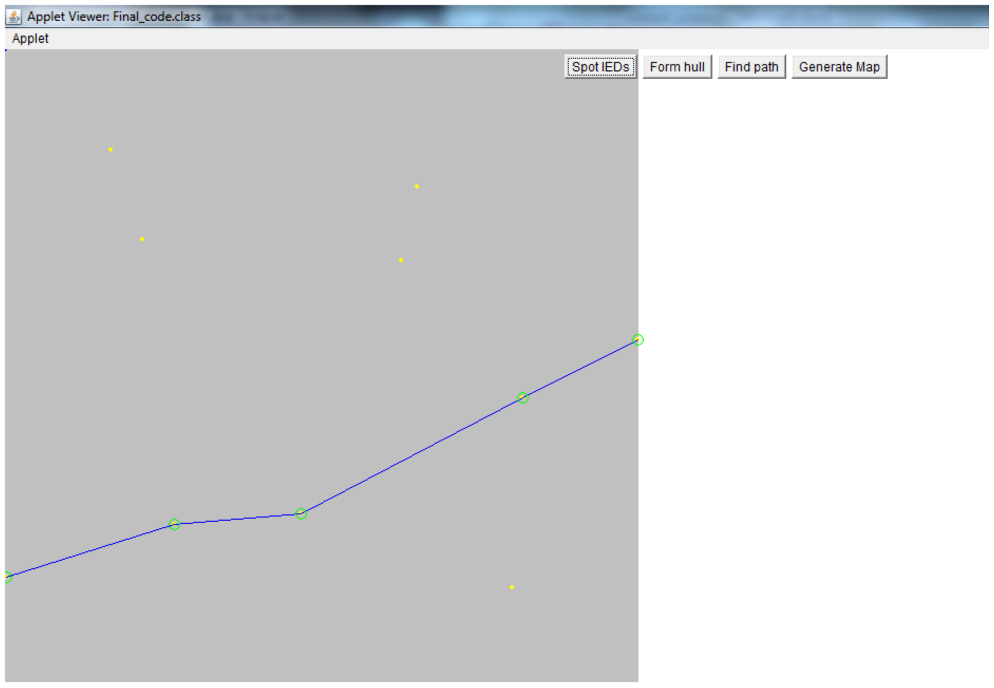

Figure 4.1. Map from source to destination of phase

The applet display in Figure 4.1 is what the commander has with him at the start of the phase, which he would use to help guide him through that phase. The yellow dots on the applet are all of the possible way points in a phase got from an instance of the OPORD corresponding to that particular phase as explained earlier. The dots circled in green are the points computed to be on the shortest path to the destination of the phase. A blue line joins these points hence providing a complete graphical interface to propagate through the phase.

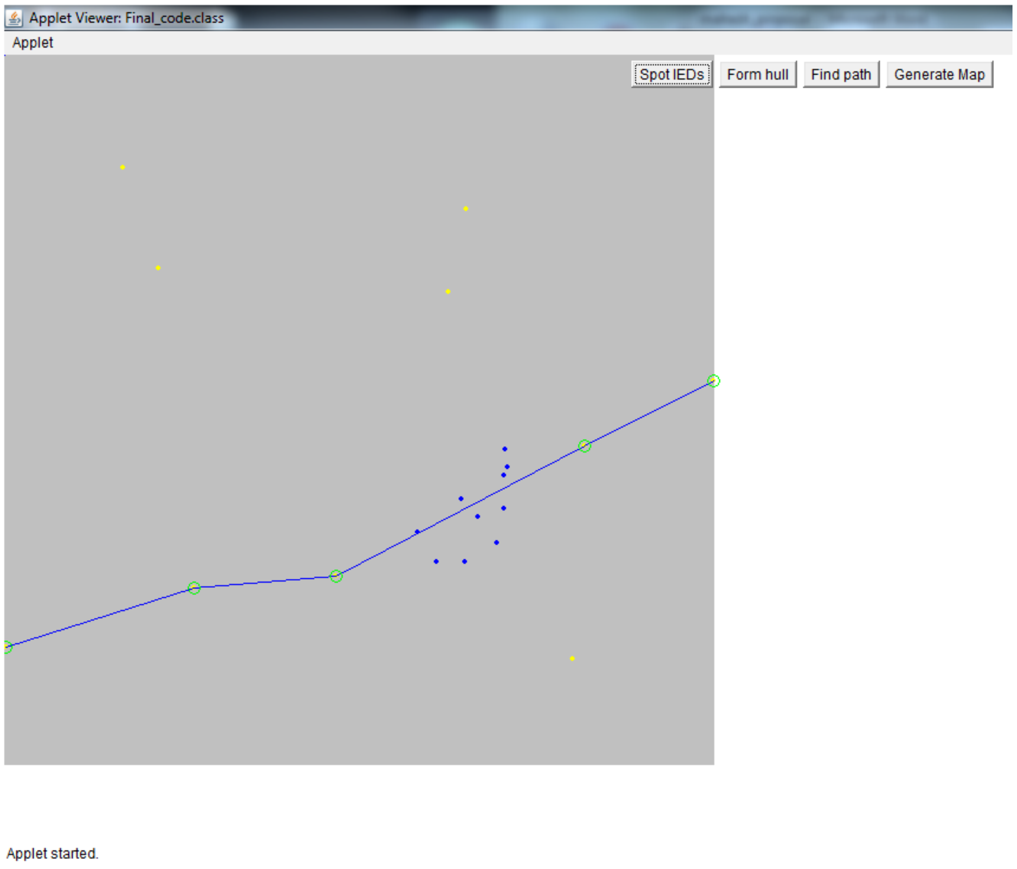

Figure 4.2. IED group spotted on the shortest path during phase execution

The commander can now start moving through the phase along the shortest path depicted by the Assessor Agent. As the commander moves through the phase, he would learn about the presence of IEDs in the area of operation of the phase and, as long as these IED do not fall on the shortest path, they should not concern the commander. If he detects a group of IEDs along the shortest path, the Assessor Agent developed here, receives the co-ordinates of the IEDs and depicts them on the applet as shown in the Figure 4.2. The blue dots in this figure represent the IEDs spotted on the shortest path.

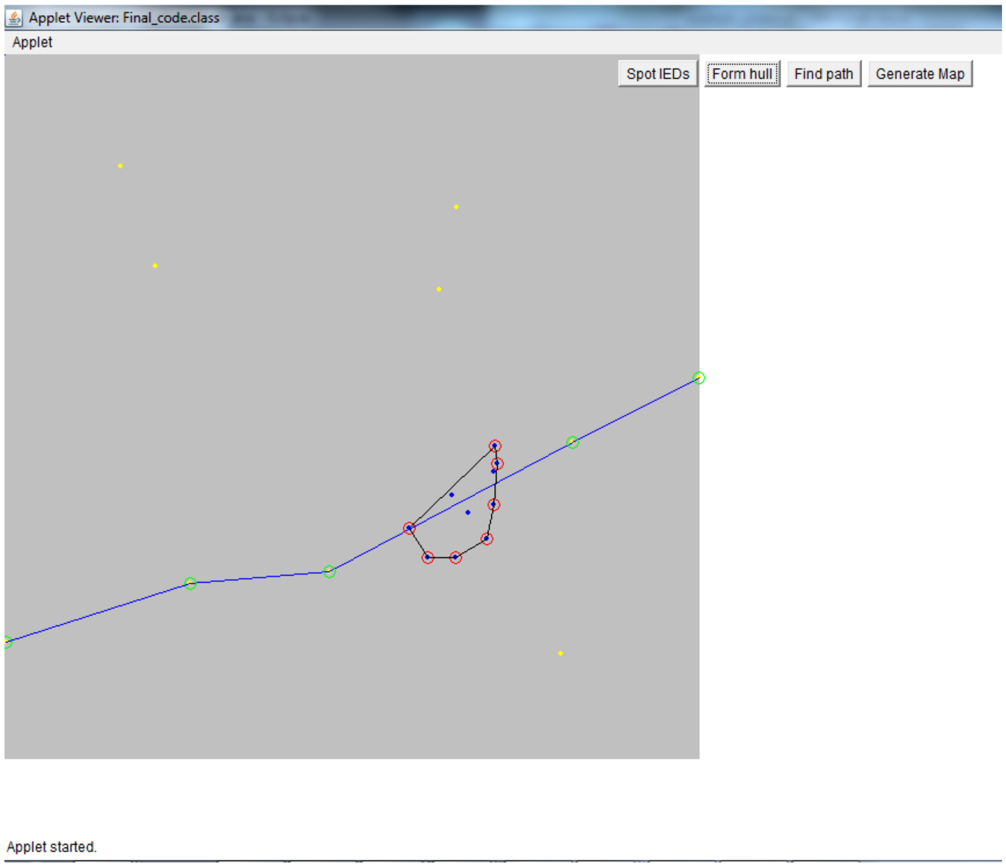

Figure 4.3. Convex hull formed around the IED group

The Assessor Agent reads the co-ordinates of these IEDs as input and forms a convex hull around the IED group clearly demarking a region of danger that the commander and his troops must avoid. The convex hull computed is then made visible to the commander by being mapped onto the applet as shown in Figure 4.3.

The points circled in red form the circumference of the danger region consisting of IEDs. Hence, the area enclosed by the black lines must be avoided at any cost by the commander. The Assessor Agent, after computing the convex hull computes the shortest path along the circumference of the convex hull as shown in Figure 4.4.

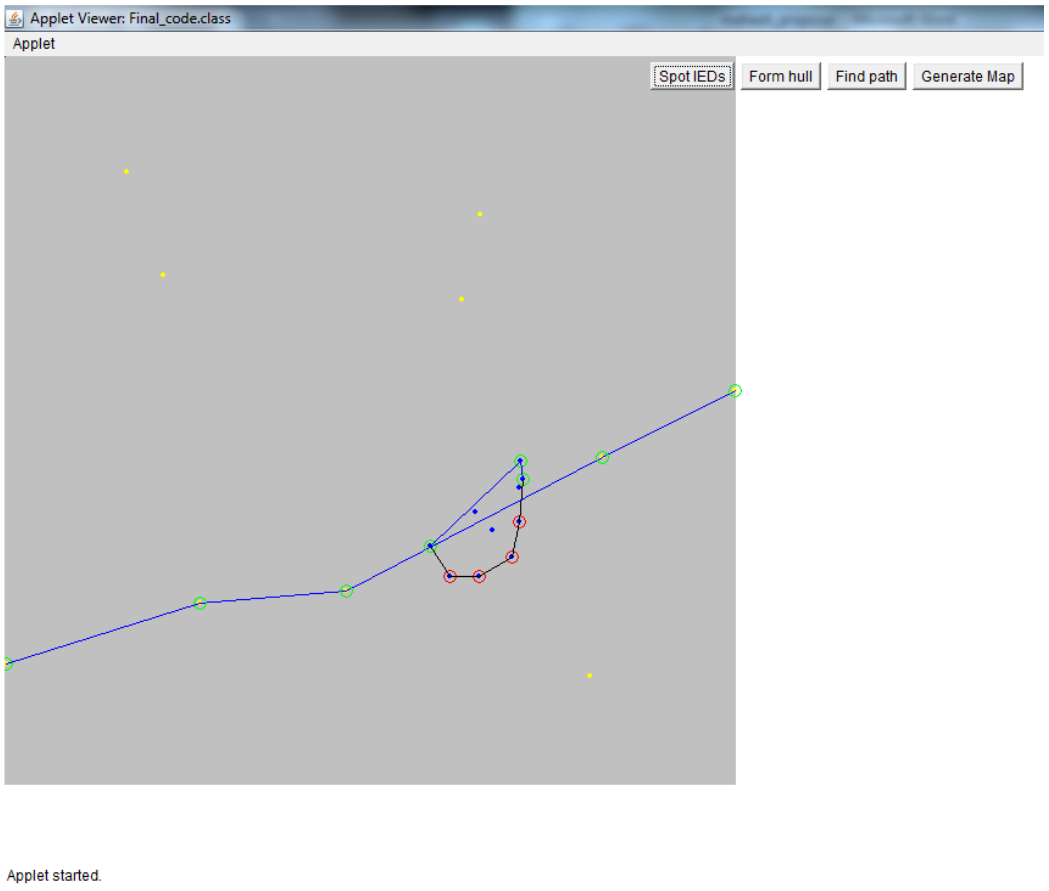

Figure 4.4. Shortest path along the convex hull computed

The blue line along the circumference of the convex hull indicates the shortest path around the hull. It is this path on the hull that becomes a part of the final route to the destination of the area of operation in the phase in progress, shown in Figure 4.5.

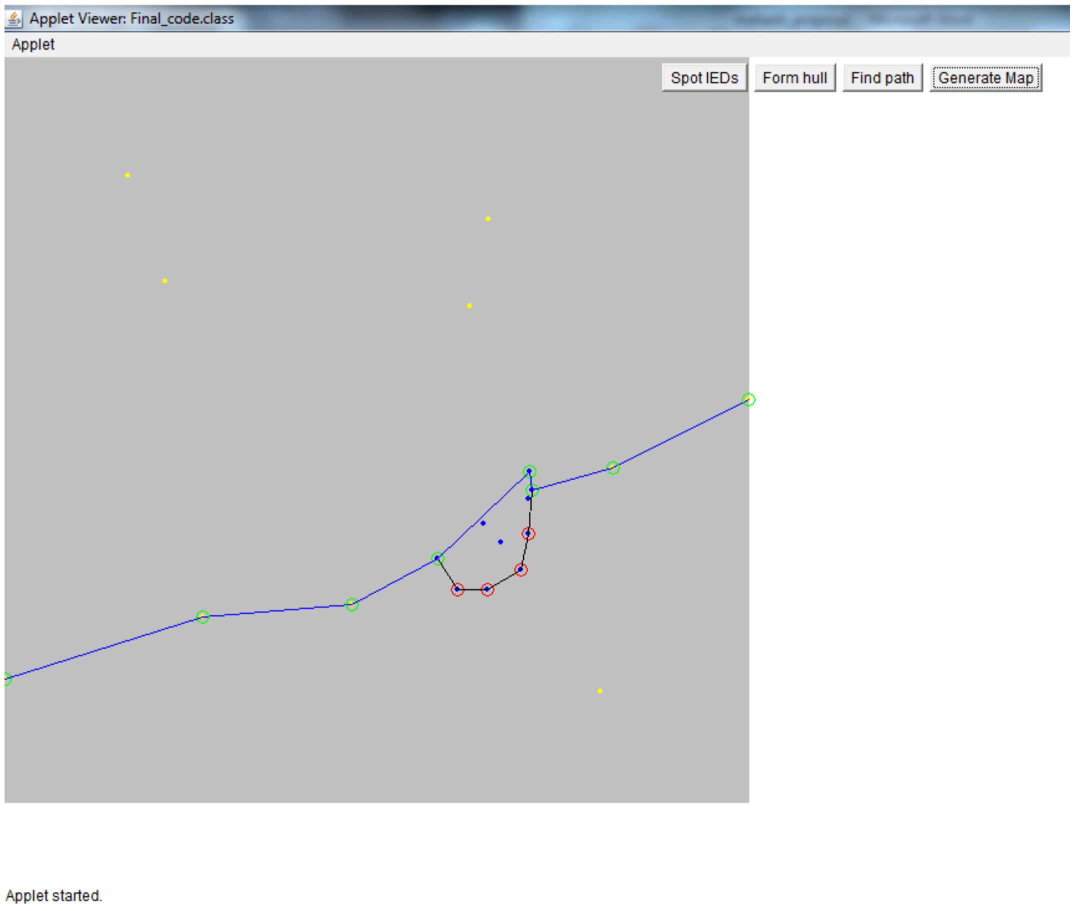

Figure 4.5. Final map with new shortest/safe path to destination

This is the map that is finally visible to the commander at the intermediate point along the initial shortest path, where he learned about the existence of an IED group. Using this newly generated map, the commander can still get to the destination of the phase in the shortest time possible but, most importantly he and his troops will get there safely.

4.3 Configuration Issues

This section describes various issues that were encountered while configuring workstations for implementation of the TITAN services from the previous phase of the development done by two previous MS students Krishnamurthy and Banda. The first issue encountered was with the deployment of the WSIG agent which failed because of conflicts in the versioning jar files in the WSIG 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. There were similar versioning issues while deploying the JMS Agent Gateway when it failed to work and some additional jar files (*beangenerator.jar* and *log4j-1.2.16.jar*) were required to configure and get things working. Another major issue was with the default display of the applet. The implementation of the Assessor Agent was done by having an image of the virtual map appearing in the first quadrant (i.e. with both x and y axes running in the positive direction)

In the applet display but the default applet displays in the fourth quadrant (i.e., with x axis running positive and the y axis running negative). The assessor agent had to be modified to suite the default display supported by the Java applet API.

CHAPTER 5

DISCUSSION

We conceptualize the tactical behavior of the TITAN sections [1] in terms of multiagent systems (MASs) and holarchies [51]. To model hierarchical units of sections, Parts/whole [57] Statecharts are used, which introduce a coordinating whole as a concurrent component on a par with the coordinated parts. Implementing a part endows an individual with the capacity for certain public behavior so that it may realize the role described by the part, while implementing a whole (in each subsumed individual, from its point of view) is taken to endow the subsumed individuals with the capacity for certain social behavior. This capacity, for a given group, is related to the technical notion of common knowledge, which is a prerequisite for coordination (among members of the given group). The whole is considered as a description abstracted from the redundant parts of a hierarchical system where this redundancy is the common knowledge shared by the team. A Parts/whole Statechart where wholes are related to common knowledge is called a *holochart*. To illustrate this conceptualization, a holochart model of a tactical behavior of a platoon of sections is developed using the Stateflow toolbox, which is an interactive simulation tool for modeling event-driven systems with Statecharts. Stateflow [59] provides the language elements required to describe hierarchical, concurrent behavior in a natural and comprehensive form. The holochart model is presented and discussed in details. Coordination and synchronization are fundamental for realizing a successful mission in which changes in tactical behavior are essential to react and adapt to the environment.

In the following section, we define the notion of common knowledge and we relate common knowledge to the concept of wholes in Parts/whole Statecharts. Section Two of this chapter describes a platoon mission. Section Three defines the holochart model, presents its properties, and explains in detail its components as well as their Stateflow implementation. Section four concludes and summarizes the holochart modeling technique.

5.1 Common Knowledge comparable with Wholes

We have already, discussed the concept of common knowledge in the background chapter. Here we relate common knowledge to the concept of Wholes. David Lewis, in his study *Convention* (1969), introduced the concept of common knowledge. Robert Aumann (1976) mathematically formulated common knowledge in a set-theoretical framework. For individuals, the need to communicate or to coordinate their behavior successfully requires common knowledge. Common knowledge is a phenomenon which underwrites much of social life. It is a fundamental form of knowledge for a multiagent system. A proposition *A* is mutual knowledge among a group of agents *G* if each agent in *G* knows that *A*. In section 2.10.1, this was formally defined and denoted by $E_G\ \varphi$, "everyone in G knows thatφ.". Mutual knowledge by itself implies nothing about what, if any, knowledge anyone attributes to anyone else. For example, suppose each student arrives for a class meeting knowing that the instructor will be late. That the instructor will be late is mutual knowledge, but each student might think only she knows the instructor will be late. However, if one of the students says openly "Peter told me he will be late again," then each student knows that each student knows that the instructor will be late, each student knows that each student knows that each student knows that the instructor will be late, and so on, *ad infinitum.* The announcement made the mutually known fact *common knowledge* among the students.

There is a distinctive link between common knowledge and Wholes. In holocharts, common knowledge is strongly tied to wholes in parts/wholes Statecharts. We could think of the whole as being a description abstracted from the redundant parts of a hierarchical system; the redundancy here is the common knowledge shared by the team. A shared situation is the arena in which the whole for a team of agents manifests itself, while the individual represents the whole by embodying it. In a broader context, the community's conventions supply the representational resources used in linguistic co-presence.

5.2 Platoon mission

Agents need to coordinate and to cooperate to perform their common task in a dynamic environment. The coordination and cooperation in multiagent systems is achieved through interaction and communication that establishes common knowledge. A mission of a platoon of sections is considered a perfect example to demonstrate the necessity to communicate and coordinate in order to form common knowledge. Thus, a platoon mission is ideal for displaying the holochart notation.

We start the mission with the platoon being divided into two "online" sections, one representing the Leader section and the other a Follower section. Each of these sections undertakes two phases of mission operation. The main objective of the platoon mission is to traverse the area of operation of a particular phase in the mission. Starting from the assembly, both the Leader and the Follower are expected to reach the destinations of their respective areas of operation during a particular phase of the mission. Also, we try to synchronize their movements between multiple phases of the mission and during the event when either Section encounter an IED group during their navigation through the particular phase of the mission. In the former case, the section that reaches the end of a particular phase waits for the other section to complete that phase as well, before each of them moves forward into the next phase of the mission together. For example, if the leader reaches the end of phase 1, it waits for the follower to complete phase 1, and then- both start phase 2 together. Coordinating the start of the second phase explains the term online. Hence, their movement between phases of a mission is synchronized.

In the latter case, where, in the middle of a phase, either section encounters an IED group on its path to the end of the area of operation of that phase, the TITAN program requires the other section at least to wait for the former to handle the event of safely getting around the IED group and then it resumes its quest to get to the end of the area of operation. For example, if during phase 1, the Leader encounters an IED group on its path to the destination, the follower waits wherever it is until the leader finishes getting around the IED group and then resumes its progress through the phase. This ensures that the end of every phase is synchronized. In a more sophisticated scenario, the follower would engage in some helpful activity, such as covering for the leader, while the leader is navigates around the IED group. The synchronization process is the key to performing the platoon mission successfully.

5.3 Platoon Holochart Model

In this section, a mission for a group of sections is modeled as a holochart simulated in Stateflow. The first level of the platoon holarchy has as its parts two sections – representing a Leader and a Follower where the coordinating whole is a concurrent component representing the common knowledge at the platoon level as shown in Figure 5.1. This analysis does not involve TBSs since the idea is to analyze the coordinated activity of individuals as specified by OPORDs and other C2 products. Note that, from the perspective of individuals in the hierarchy, the commander, or the commander along with the TBS, is another individual. The second level of the holarchy consists of two XOR states for each participating section (Leader and Follower), each representing separate phases of the mission that the sections has to progress through. Holocharts restrict the direct interaction between the parts (in this case the 2 sections – Leader and Follower), which results in encapsulated portable entities. If a part is to be added, for example, another TBS, only the platoon whole needs to be modified to reflect the addition.

In Figure 5.1, the analysis has communication between the sections mediated by the whole. The whole, however, is not an additional individual; it simply represents the coordination among the parts making up the overall unit. In a more involved scenario, the sections would have transitions that depend on, for example, events in the environment. Such transitions would not be reflected in the whole since they do not involve coordination. The TITAN whole essentially represents the common knowledge required for the sections to coordinate.

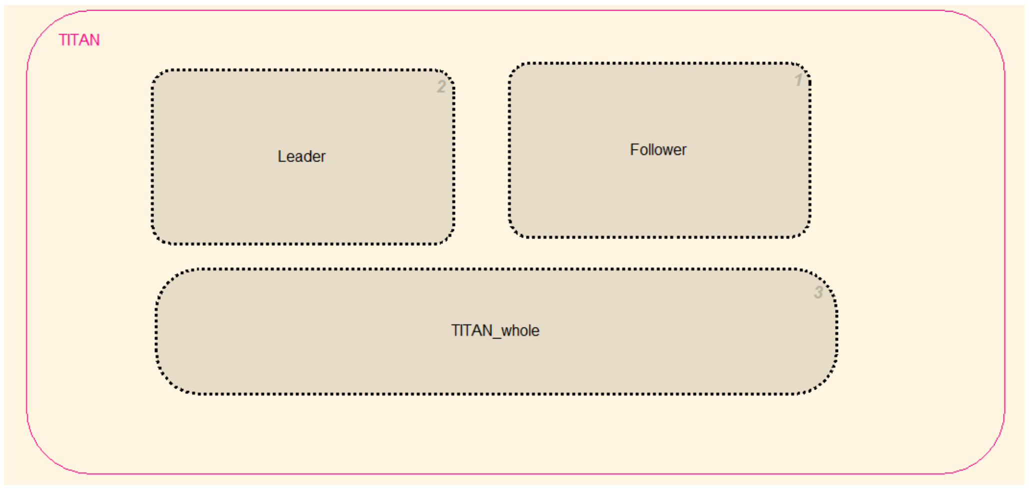

Figure 5.1. Holochart of a platoon of sections

The platoon mission modeled as a holochart can be explained as follows. The external events and the mission initialization signal are controlled by and simulated with the help of manual links in Simulink as seen in Figure 5.2. The initialization signal here refers to the “advance” signal which causes the leader to start its first phase of the mission which in turn starts the first phase of the follower. The other four external events controlled by the user refer to the event of encountering an IED group by each section in each of their two phases.

As stated earlier, it is the “advance” signal shown in the figure that starts off the simulation by causing the first phase of the leader to start. The four IED group signals simulate the environment of IEDs existing in the area of operation of a phase. These IED groups need to be handled by the section in order for them to progress through the mission. The following section describes the first level of the platoon holarchy starting with the “TITAN_whole” and then the “TITAN_part”.

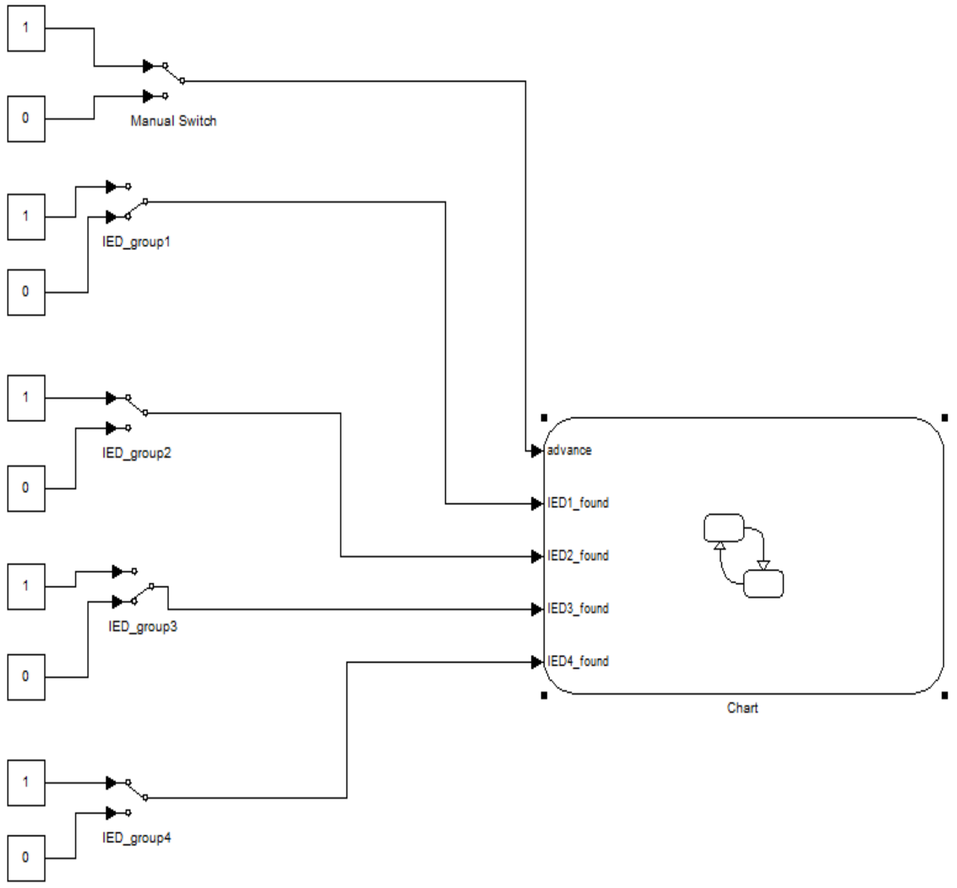

Figure 5.2. Initialization signal and the 4 external events

5.3.1 TITAN Whole. The TITAN Whole state is orthogonal to the Leader and Follower states. It is this whole that represents the co-ordination of the two sections. The TITAN whole is an XOR state with three sub-states, as shown in Figure 5.3.

It is the “TITAN_start” state that triggers the start of the simulation when it receives the positive edge of the “advance” signal. This initialization event along with causing a state change to “TITAN_in_progress” peforms the action “Start_leader” which causes the Leader to start its journey through the mission. It is the Leader that starts the Follower and both execute their activities in parallel. The “TITAN_in_progress” state is solely responsible for the co-ordination between the two sections, as shown in Figure 5.4. It is basically composed of two XOR states, each corresponding to the co-ordination within a particular phase of the TITAN mission.

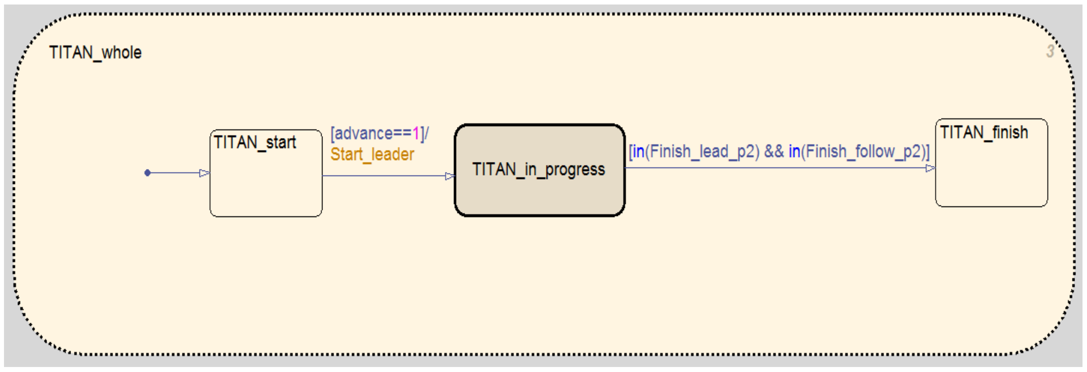

Figure 5.3. Statechart of the TITAN whole

The state that represents each phase in Figure 5.4 is in turn composed of four states that handle the co-ordination between the two sections. Also, the two states “Phase1” and “Phase2” are conceptually similar. Consider how the “Phase1” state reflects the coordination between the two sections through the first phase of the mission. As we know, the TITAN program requires that when either TBS encounters an IED group, the other TBS must wait for the former to complete handling this event before it resumes its journey through the phase. This aspect is realized during “Phase1” by the states “L1_wait” and “F1_wait”.

The state “L1_wait” is reached when the Follower encounters an IED group (triggered by the event “wait_L1” issued by the follower) on its path of the area of operation. The entry to this state performs an action “L1_stop” (see Figure 5.4) which causes the Leader to go into a Wait state as shown in Figure 5.6. The control exits this state on the occurrence of the event “Finish_L1” that is triggered by the Follower in phase1 on completion of handling the IED group successfully. This “Finish_L1” event issued by the Follower triggers the action “Resume_L1” in the Leader, commanding the Leader to resume its journey through the area of operation (see Figure 5.6). Similar activities take place with the Leader encountering an IED group and entering the “F1_wait” state.

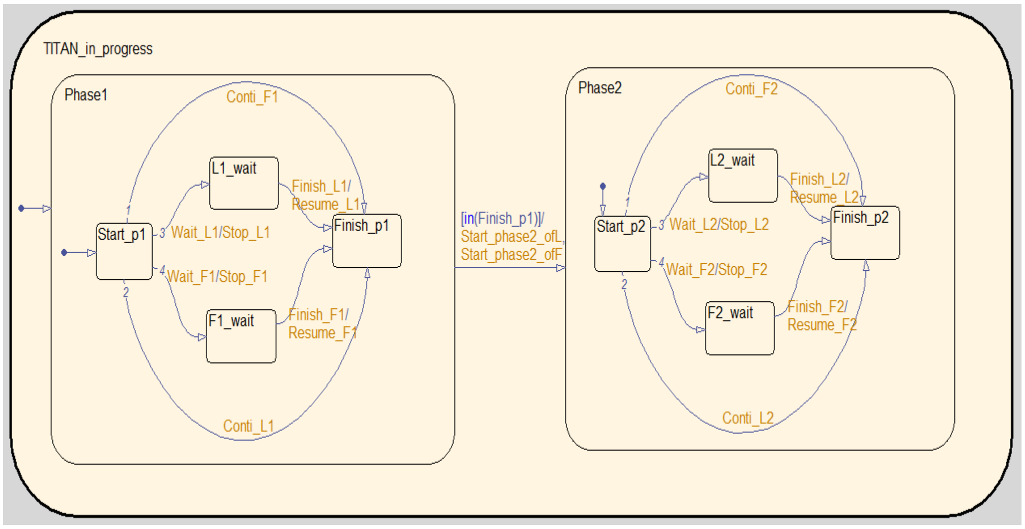

Figure 5.4. Statechart of “TITAN_in_progress” - state in charge of co-ordination

In the event of either section not encountering an IED group and navigating straight through to the end of the area of operation, the events “Conti_F1” and “Conti_L1” cause the Follower and the Leader respectively to get to the end of phase 1 indicated by the state “Finish_p1”. The control exits “Finish_p1” only after both the Leader and Follower have reached the end of phase1.

It is only after the completion of phase 1 that phase 2 is started which triggers two actions “Start_phase2_pfL” and “Start_phase2_pfF” which cause the Leader and

Follower respectively to start their activities in propagating through phase 2 of the mission. Te state "Phase2" in Figure 5.4 co-ordinates the activities between the Leader and Follower through the second phase just as the way "Phase1" state did for the first phase of the mission.

5.3.2 TITAN Part. The Leader and Follower sections are considered as the parts of the TITAN platoon model and both are modeled similarly. The coordination between their activities through multiple phases (in this case two) is represented by the "TITAN_whole" state as was discussed in the previous section. We first consider how the Leader is modeled.

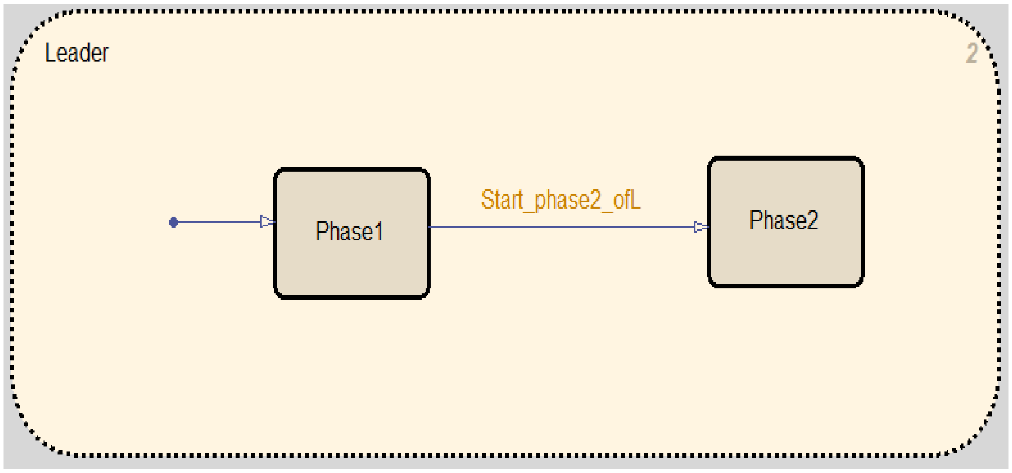

Figure 5.5. Statechart of the Leader of a Part of the TITAN Holarchy

As we can see from Figure 5.5, the Leader is an XOR state with two sub-states, each representing a separate phase of the mission. The state"Phase1" in Figure 5.5 traversing through the area of operation of phase1 on receiving the event "Start_leader", which in turn triggers the action "Start_follower" (causing the Follower to start its activates during phase1) both of which are seen in Figure 5.6. The "Start_leader" event is triggered as a consequence of the initialization event "advance", as seen in Figure 5.3, and results in a transitition to the state "Lead_p1_in_progress", which, as the name suggests indicates that the Leader is in the state of progressing through phase1 (see Figure 5.6).

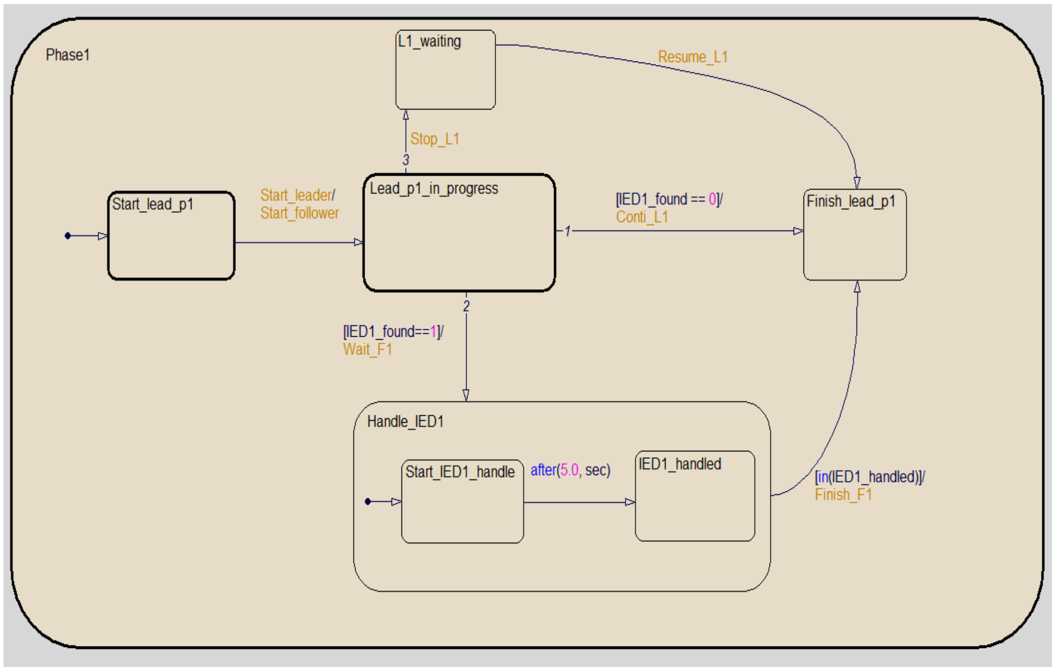

Figure 5.6. Statechart for a particular phase of a section

In the event of the Leader encountering an IED field (triggered by external event "IED1_found"), the Leader transitions to "Handle_IED1", which, as the name suggests, tries to handle the IED as shown in Figure 5.6. As a result of this event, an action "Wait_F1" is issued that orders the Follower to wait until the Leader completes handling the IED field. The handling of the IED field here is simulated as a delay and, when the delay time elapses the phase is completed, represented by a transition to state

“Finish_lead_p1”. This transition also triggers an action “Finish_F1” which commands the Follower to resume its pursuit through the phase.

In the case of the Follower encountering an IED field, the Leader receives a “Stop_L1” event that forces the leader into the “L1_Waiting” state where it waits until the Follower has completed the handling of the IED field that it encountered. This completion of IED handling is intimated to the Leader through the action “Resume_L1” as shown in Figure 5.6.

5.4 Conclusion

A holochart is a Parts/whole Statechart where a whole captures the common knowledge of the group consisting of the other sub-states (“parts”) in the AND state of which the whole is a sub-state. To illustrate this conceptualization, a holochart model of the tactical behavior of a platoon made up of two sections is developed using the Stateflow toolbox, which is an interactive simulation tool for modeling event-driven systems with Statecharts. The platoon’s mission is presented and the implementation of the holochart model is discussed in detail. TITAN_Whole, Leader, and Follower are orthogonal components of the model. Each section Leader and Follower has two phases. The relation between wholes and common knowledge is discussed as well. Our holochart model is well organized and encapsulated. The holochart model allows a better understanding of the tactical behavior of the platoon of sections. The holochart model can be easily expanded to include additional environment modeling and complete reactive behavior of the platoon.

CHAPTER 6

CONCLUSION AND FUTURE WORK

The Information Dissemination and Management (ID&M) services of the TITAN program are required to use intelligent-agent software frameworks because the tactical command and control domain is suitable for the application of this technology in the sense that, intelligent agents 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 even be immediately realized by war-fighters engaged in battle. Three important information technologies namely: distributed event based system, multiagent systems, and Web services are incorporated for collaboration and dissemination of information.

Functional stubs of three important TITAN ID&M services were implemented earlier in the project. The Alert and Warning Service (AWS) monitors mission progress and raises alerts and warnings. The OPORD Support Service (OPS) works on command and control documents in the army's common information exchange data model. Finally, the Workflow Orchestration Support (WOS) service constructs and executes workflows from tasks.

The work reported here extends the functionality of the AWS service to provide for a completely informative graphical interface with the shortest safe path through the area of operation of a particular phase in the TITAN mission. This would serve the commander as an electronic/virtual map and guide him/her through the respective phase. This implementation also introduces synchronization into the coordination of multiple troops/commanders within a single phase of the TITAN mission. The distance of the shortest path through the area of operation of a phase is computed here with the help of distance formula, and if the average speed at which the respective troop/commander would navigate through a phase is known, the event of all of the participating troops/commanders reaching the end of the phase is synchronized.

Several extensive issues are addressed by this research and one such issue is the coordination of hierarchically structured TITAN units. The concept of common knowledge is a prerequisite for such coordination and is strongly tied to wholes in the Parts/whole Statecharts notation. We can now think of the whole as being a description abstracted from the redundant parts of a hierarchical system; the redundancy here is the common knowledge shared by the teams. This conceptualization of common knowledge being comparable to Wholes in Parts/whole Statecharts is realized as a simulation with the help of the Stateflow toolbox.

Future work in this project would include implementing the rest of the TITAN services, hence making the TBSs, supported by all these services, more dynamic and versatile. New technologies can be used to standardize asynchronous push abilities of Web service to make the architecture unbounded and flexible. An extension pertaining to the AWS service implemented in this project would be to develop a tablet/smartphone application to make everything we saw in the implementation section, available on a tablet/smartphone. The output of such a hand held device application can quiet literally serve as a virtual/electronic map, displaying the shortest/safe path through an area of operation of a phase that the particular troop/commander can use as a guide.

REFERENCES

1. US Army. (2008). *A Guide for Developing Agent-Based Services, Revision 14.0.*
2. Bellifemine, F., Caire, G., Greenwood, D. (2007). *Developing multi-agent systems with JADE*. John Wiley & Sons, Inc. Hoboken, NJ.
3. Huhns, M & Singh, M. (2005). *Service-Oriented Computing: Semantics, Processes, Agents*. Wiley.
4. Monson-Haefel, R. & Chappell, D. A. (2001). *Java Message Service*, O'Reilly.
5. Cougaar New User Road Map, Retrieved August 9, 2011 from http://cougaar.cougaar.org/software/latest/doc/roadmap.html.
6. Mühl, G., Fiege, L., Pietzuch, P. (2006). *Distributed Event-Based Systems,* Springer, Berlin.
7. Carzaniga, A. & Fenkam, P. (2004). *Third International Workshop on Distributed Event- Based Systems*: DEBS '04 (workshop overview). In Proceedings of the 26th International Conference on Software Engineering, 750-751.
8. Fiege, L., Mühl, G., Gärtner, F. C. (2002). *Modular event-based systems, The Knowledge Engineering Review*, 17(4), 359-388.
9. Carzaniga, A., Di Nitto, E., Rosenblum, D. S. & Wolf, A. L. (1998). *Issues in supporting event-based architectural styles*. In Proceedings of the Third International Workshop on Software Architecture *(ISAW '98*). 17–20.
10. Sjöberg, P. (2007). The *Java Message service 1.0.2*. Retrieved on August 10, 2011 from http://www.cs.helsinki.fi/u/campa/teaching/j2me/pa.

11. The World Wide Web Consortium. *W3C: World Wide Web Consortium.* Retrieved, September 20, 2011 from http://www.w3.org/.

12. Organization for the Advancement of Structured Information Standards. *About OASIS*. Retrieved, September 20, 2011 from http://www.oasis-open.org/who/.

13. MDM, *MDM Web Services*, Retrieved on November 27, 2011 from http://help.sap.com/saphelp_mdm71/helpdata/en/45/018c03166a0486e10000000a155369/content.htm .

14. The World Wide Web Consortium. *W3C. Web Services Glossary*. Retrieved on September 22, 2011, from http://www.w3.org/TR/2004/NOTE-ws-gloss-20040211/.

15. The World Wide Web Consortium. *W3C. Simple Object Access Protocol (SOAP)* Retrieved September 10, 2011, from http://www.w3.org/TR/2000/NOTE-SOAP-20000508/.

16. OASIS. *UDDI Version 3.0.2*. Retrieved October 21, 2011 from http://www.uddi.org/pubs/uddi_v3.htm.

17. *"jUDDI." Apache Web Services Project*. Retrieved July 25, 2011 from http://ws.apache.org/juddi/.

18. Ort, E. *Java Architecture for XML Binding*. Retrieved July 30, 2011, from http://www.oracle.com/technetwork/articles/javase/index-140168.html.

19. Banda, S. (2011). *Coordination of Hierarchical Command and Control Services*, MS Thesis, North Carolina A&T State University, Greensboro, NC.

20. ORACLE docs. *Using JAXB Data Binding: Standard Data Type Mapping*, Retrieved on October 17, 2011 from http://docs.oracle.com/cd/E12840_01/wls/docs103/webserv/data_types.html#wp223908.

21. Wooldridge, M. (2002). *An Introduction to Multiagent Systems*. John Wiley & Sons, Chichester, England.

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

23. 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.

24. Hoare, C. A. R. (1978). Communicating sequential processes. *Communications of the ACM*, 21, 666-677.

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

26. The Foundation for Intelligent Physical Agents. *Welcome to FIPA*. Retrieved July 27, 2011, from http://www.fipa.org.

27. The Foundation for Intelligent Physical Agents. *How to Become a FIPA Member*. Retrieved July 27, 2011, from http://www.fipa.org/subgroups/.

28. The JADE-Board, *JADE: Java Agent Development Framework*. Retrieved July 27, 2011 from http://jade.tilab.com/.

29. Ernest Friedman-Hill. (2003). *Jess in Action: Rule-Based Systems in Java*. Manning Publications Company, Greenwich, CT.

30. Curry, E., Chambers, D., Lyons, G. (2004). *Enterprise service facilitation within agent environments*. In Proceedings of the *IASTED* Conference on Software Engineering and Applications, November 9-11, 2004, MIT, Cambridge, MA, USA.

31. *JADE Web Services Integration Gateway (WSIG) GUIDE, Telecom Italia*, (2008). Retrieved July 27, 2011 from http://jade.tilab.com/doc/tutorials/WSIG_Guide.pdf.

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

33. Kitchin, D., Quark, A., Cook, W., & Misra, J. (2009). *The Orc Programming Language.*

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

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

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

37. 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.

38. 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.

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

40. 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.

41. 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).*

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

43. 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.

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

45. 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.

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

47. 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.

48. *Being There: Putting Brain, Body, and World Together Again*: The MIT Press.

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

50. 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, 101,* 81-88. Springer Berlin / Heidelberg.

51. Koestler, A. (1968). *The Ghost in the Machine*. Macmillan, New York.

52. Wooldridge, M. (1999). *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*. *G. Weiss, ed.,* The MIT Press, Cambridge, Massachusetts, 619 pages.

53. Giret, A. & Botti, V. (2004). Holons and Agents. *Journal of Intelligent Manufacturing, 15,* 645 – 659.

54. Kaye, J. & Castillo, D. (2002). Describing Behaviors Using Statecharts in Flash MX for Interactive Simulation: *How to Construct & Use Device Simulations Delmar Learning*, pp. 75 – 102.

55. Harel, D. (1987). Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming, 8*, 231-274.

56. D. Harel et al. (1987). On the formal semantics of statecharts. *Proc. Symp. on Logic in Comp. Science (LICS '87),* IEEE Computer Society, 54-64.

57. Pazzi, L. (1997). Extending Statecharts for Representing Parts and Wholes. *Proc. EUROMICRO 97 Conf.*, pp. 207-214, Budapest.

58. Pazzi, L. (1999). Implicit versus explicit characterization of complex entities and events. *Data & Knowledge Eng., 31(2),* 115-134.

59. Hamon, G., & Rushby, J. *An Operational Semantics for Stateflow*, Retrieved on July 12, 2011 from http://www.csl.sri.com/users/rushby/papers/fase04.pdf.

60. Data Management Working Group (DMWG), Great Britain (GBR), *JC3IEDM Overview*, Edition 3.1b, Greding Germany, 13 December 2007.

61. Headquarters, Department of the Army. (2003). *Field Manual 6.0*, Chapter 6,Washington, DC. Retrieved on January 20, 2011, from http://www.globalsecurity.org/military/library/policy/army/fm/6-0/chap6.htm#6-2

62. Headquarters, Department of the Army. (1997). *Field Manual No. 101-5 (FM 101-5)*, Washington, DC, 31 May 1997.

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

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

65. CERDEC, US Army. (2008). *OPORD Support Service Agent Architecture Specification, Revision 1.5*.

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

APPENDIX

AWS Assessor Agent

```
import java.applet.Applet;

import java.awt.*;

import java.awt.event.*;

import java.awt.Button;

import java.awt.Graphics;

import java.awt.Color;

import java.awt.Dimension;

import java.util.ArrayList;

import java.util.Collections;

import java.util.Comparator;

import java.util.Iterator;

import java.util.Random;


import java.util.PriorityQueue;

import java.util.List;

import java.util.regex.Matcher;

import java.util.regex.Pattern;


import java.lang.Math;


class Vertex implements Comparable<Vertex>

{
    public String name;
    public Edge[] adjacencies;
    public double minDistance = Double.POSITIVE_INFINITY;
    public Vertex previous;
    public Vertex(String argName) { name = argName; }
    public String toString() { return name; }
    public int compareTo(Vertex other)
    {
        return Double.compare(minDistance, other.minDistance);
    }
}

class Edge
{
    public Vertex target;
    public double weight;
    public Edge(Vertex argTarget, double argWeight)
    { target = argTarget; weight = argWeight; }
}

public class Final_code extends Applet implements ActionListener
{

            //Dijistras algos declarations

            int dNum=10;
            int xdijPoints[];
    int ydijPoints[];
    int shortest_path[];

    int xdijpathPoints[];
    int ydijpathPoints[];

    int size_of_path_list;

    //Convex hull algos declarations

    Random rnd;
    int pNum = 10;

    //These are the random land mines generated
    int xPoints[];
    int yPoints[];
  //These contain the hull points
    int xPoints2[];
    int yPoints2[];

    //Arrays for computing path around the hull
    //Points along 1 path
    int xhullpath1[];
    int yhullpath1[];
    //Points along the other path
```

```
    int xhullpath2[];
    int yhullpath2[];

    //Final selected path
    int xfinalhullpath[];
    int yfinalhullpath[];


    int num;
    int p;
    int w, h;

    int hull_path_count;

    int pos1,pos2;

    //Some paramenter that need to be entered by the commander on encoutering the IED
fields
    int  xinit,yinit,xend,yend;
    int xhullinit,yhullinit,xhullend,yhullend;

    Button bt1,bt2,bt3,bt4;


//    Button bt;
```

```
public void init()
{
        //Dijistras algos initialization

        xdijPoints = new int[dNum];
        ydijPoints = new int[dNum];
        shortest_path = new int[dNum];

        xdijpathPoints = new int[dNum];
    ydijpathPoints = new int[dNum];



    //Convex hull algos initialization

    Dimension size = getSize();
        w = size.width;
        h = size.height;
        System.out.println(w + "\n" + h + "\n");
        rnd = new Random();

        //These are the random land mines generated
        xPoints = new int[pNum];
        yPoints = new int[pNum];
        num = 0;
        //These contain the hull points
        xPoints2 = new int[pNum];
```

```
yPoints2 = new int[pNum];

//Points along path1 on the hull
xhullpath1 = new int[pNum];
yhullpath1 = new int[pNum];
//Points along path2 on the hull
xhullpath2 = new int[pNum];
yhullpath2 = new int[pNum];

//Final path on the hull
xfinalhullpath = new int[pNum];
yfinalhullpath = new int[pNum];

hull_path_count=0;

p=330;

dijistra();

bt1 = new Button("Spot IEDs");
bt1.addActionListener(this);
add(bt1);

bt2 = new Button("Form hull");
bt2.addActionListener(this);
add(bt2);
```

```
        bt3 = new Button("Find path");
        bt3.addActionListener(this);
        add(bt3);

        bt4 = new Button("Generate Map");
        bt4.addActionListener(this);
        add(bt4);

}

public void actionPerformed(ActionEvent ev)
{
        if ( ev.getSource() == bt1 )
                {
                ide_generate(p);
                }
                repaint();
        if ( ev.getSource() == bt2 )
                {
                        quickconvexhull();
                }
                repaint();
                if ( ev.getSource() == bt3 )
                {
                        compute_hull_path();
                }
                repaint();
                repaint();
                if ( ev.getSource() == bt4 )
                {
                        generate_map();
                }
                repaint();
    }

    public void generate_map()
    {
        //Some parameters to be entered by user as well as some display config to
generate the map
        xinit=280;yinit=440;
        xend=490;yend=330;
        xhullinit=xPoints2[0];yhullinit=yPoints2[0];
        xhullend=xPoints2[pos2];yhullend=yPoints2[pos2];
    }

//Dijistras algorighm implementation code

    public static void computePaths(Vertex source)
    {
      source.minDistance = 0.;
      PriorityQueue<Vertex> vertexQueue = new PriorityQueue<Vertex>();
    vertexQueue.add(source);

    while (!vertexQueue.isEmpty())
    {
        Vertex u = vertexQueue.poll();

        // Visit each edge exiting u
        for (Edge e : u.adjacencies)
        {
            Vertex v = e.target;
            double weight = e.weight;
            double distanceThroughU = u.minDistance + weight;

            if (distanceThroughU < v.minDistance)
            {
                vertexQueue.remove(v);
                v.minDistance = distanceThroughU ;
                v.previous = u;
                vertexQueue.add(v);
            }
        }
    }
}

public static List<Vertex> getShortestPathTo(Vertex target)
{
        List<Vertex> path = new ArrayList<Vertex>();
        for (Vertex vertex = target; vertex != null; vertex = vertex.previous)
            path.add(vertex);
        Collections.reverse(path);
        return path;
    }


    public void dijistra()
    {
            xdijPoints[0] = 1; xdijPoints[1] = 160; xdijPoints[2] = 100; xdijPoints[3] = 130;
xdijPoints[4] = 280; xdijPoints[5] = 490; xdijPoints[6] = 480; xdijPoints[7] = 375;
xdijPoints[8] = 390; xdijPoints[9] = 599;
            //{0,160,100,130,280,400,480,375,390,600};
            ydijPoints[0] = 500; ydijPoints[1] = 450; ydijPoints[2] = 95; ydijPoints[3] = 180;
ydijPoints[4] = 440; ydijPoints[5] = 330; ydijPoints[6] = 510; ydijPoints[7] = 200;
ydijPoints[8] = 130; ydijPoints[9] = 275;
            Vertex v0 = new Vertex("v0");
                        Vertex v1 = new Vertex("v1");
                        Vertex v2 = new Vertex("v2");
                        Vertex v3 = new Vertex("v3");
                        Vertex v4 = new Vertex("v4");
                        Vertex v5 = new Vertex("v5");
                        Vertex v6 = new Vertex("v6");
                        Vertex v7 = new Vertex("v7");
                        Vertex v8 = new Vertex("v8");
                        Vertex v9 = new Vertex("v9");
```

```
                    v0.adjacencies = new Edge[]{ new Edge(v1, 5),
                              new Edge(v2, 9)};
                    v1.adjacencies = new Edge[]{ new Edge(v4, 4)};
                    v2.adjacencies = new Edge[]{ new Edge(v3, 3),
                            new Edge(v4, 10) };
                    v3.adjacencies = new Edge[]{ new Edge(v7, 7),
                                                                new Edge(v8, 6)};
                    v4.adjacencies = new Edge[]{ new Edge(v6, 8),
                                                                new Edge(v5, 5),
                                                                new Edge(v7, 6)};
                    v5.adjacencies = new Edge[]{ new Edge(v9, 6)};
                    v6.adjacencies = new Edge[]{ new Edge(v5, 4)};
                    v7.adjacencies = new Edge[]{ new Edge(v9, 7)};
                    v8.adjacencies = new Edge[]{ new Edge(v9, 7)};
                    v9.adjacencies = new Edge[]{ new Edge(v6, 6)};

                    //Vertex[] vertices = { v0, v1, v2, v3, v4, v5, v6, v7, v8 ,v9 };

                    computePaths(v0);

                    System.out.println("Distance to " + v9 + ": " + v9.minDistance);
                    List<Vertex> path = getShortestPathTo(v9);
                    System.out.println("Path: " + path);

                    //Find the size of the list that has the shortest path (number of points on
shortest path)
```

```
size_of_path_list = path.size();
System.out.println("No. of nodes in path = " + size_of_path_list);

//Regular expression for capturing the nodes and create an array out of
them
Pattern node_on_shortert_path = Pattern.compile("v([0-9]*)");

//Capture the integer group and push them onto shortest_path[]
for (int j=0;j<size_of_path_list;j++)
{
        Vertex temp = path.get(j);
        String s= temp.name;

        Matcher m = node_on_shortert_path.matcher(s);
        if (m.matches())
        {
                shortest_path[j]= Integer.parseInt(m.group(1));
        }
}

//Setting the shortest path x&y nodes
for(int j=0; j<size_of_path_list;j++)
{
        xdijpathPoints[j] = xdijPoints[shortest_path[j]];
        ydijpathPoints[j] = ydijPoints[shortest_path[j]];
        //System.out.println("intiger " + shortest_path[j] + "\t" );
```

```
                    }
                    /*for (int j=0;j<size_of_path_list;j++)
                    {
                            shortest_path[j]= path.get(j);
                    }*/

    }

//Convex hull algorithm implementation code

  //Generates the IDEs (here randomly generated)
    public void ide_generate(int p1)
    {
            // random
            for ( int i = 0; i < pNum; i++ )
            {
                    xPoints[i] = p1 + rnd.nextInt(w-100);
                    yPoints[i] = p1+ rnd.nextInt(h-100);
            }
    }

    //Computes the optimal/shortest across the convex hull after it is formed
    public void compute_hull_path ()
    {
            //pos1=leftmost point (not required)
            double min1=999,max1=0;
```

```
for(int i=0;i<num;i++)
{
        if(xPoints2[i] < min1)
        {
                min1=xPoints2[i];
                pos1=i;
        }
}
//pos2=right most point
for(int i=0;i<num;i++)
{
        if(xPoints2[i] > max1)
        {
                max1=xPoints2[i];
                pos2=i;
        }
}

//Group1 of points on the hull (between pos1 and pos2 : here pos1=0)
//changed <=pos2+1 to pos2
for (int i=0;i<=pos2;i++)
{
        xhullpath1[i] = xPoints2[i];
        yhullpath1[i] = yPoints2[i];
}
```

```
//System.out.println("The value of i after group1 completed:" + blah+ "\n");
//cnt is the no. of points in group2
System.out.println("num="+ num + "\n");

//changed cnt - removed the +1
int cnt=(num-pos2);

//Group2 of points on the hull
for (int i=0;i<=cnt-1;i++)
{
        xhullpath2[i] = xPoints2[pos2+i];
        yhullpath2[i] = yPoints2[pos2+i];
}
//Add the initial point as well for computation added to Group2
//changed - cnt+1 on both to cnt
xhullpath2[cnt]=xPoints2[0];
yhullpath2[cnt]=yPoints2[0];

//Reversing group2
int temp1,temp2;
for(int j = 0; j<((cnt+1)/2); j++)
{
        //Reversing x-points
        temp1 = xhullpath2[j];
        xhullpath2[j] = xhullpath2[cnt-j];
        xhullpath2[cnt-j] = temp1;
```

```
            //Reversing y-points
            temp2 = yhullpath2[j];
            yhullpath2[j] = yhullpath2[cnt-j];
            yhullpath2[cnt-j] = temp2;
        }

        //Distance along path1 (Group1 points)
        double dist, sum_group1 = 0, sum_group2 = 0;
        for(int i=0;i<=pos2;i++)
        {
            dist=0;
            dist = Math.sqrt(((xhullpath1[i]-xhullpath1[i+1])*(xhullpath1[i]-
    xhullpath1[i+1])) + ((yhullpath1[i]-yhullpath1[i+1])*(yhullpath1[i]-yhullpath1[i+1])));
            sum_group1 = sum_group1 + dist;
        }

        //Distance along path2 (Group2 points)
        double dist1;
        for(int i=0;i<=cnt;i++)
        {
            dist1=0;
            dist1 = Math.sqrt(((xhullpath2[i]-xhullpath2[i+1])*(xhullpath2[i]-
    xhullpath2[i+1])) + ((yhullpath2[i]-yhullpath2[i+1])*(yhullpath2[i]-yhullpath2[i+1])));
            sum_group2 = sum_group2 + dist1;
        }
```

```
          // Setting the final shortest path along the hull
          if(sum_group1<sum_group2)
          {
                    hull_path_count=pos2+1;
                    for (int i=0;i<=pos2;i++)
          {
                    xfinalhullpath[i]=xhullpath1[i];
                    yfinalhullpath[i]=yhullpath1[i];
          }
          }
          else
          {
                    hull_path_count=cnt+1;
                    for (int i=0;i<=cnt;i++)
          {
                              xfinalhullpath[i]=xhullpath2[i];
                              yfinalhullpath[i]=yhullpath2[i];
          }
          }
}

// check whether point p is right of line ab
public int right(int a, int b, int p)
{
        return (xPoints[a] - xPoints[b])*(yPoints[p] - yPoints[b]) - (xPoints[p] -
xPoints[b])*(yPoints[a] - yPoints[b]);

    }


    // square distance point p to line ab

    public float distance(int a, int b, int p)

    {

        float x, y, u;

        u = (((float)xPoints[p] - (float)xPoints[a])*((float)xPoints[b] - (float)xPoints[a]) +
((float)yPoints[p] - (float)yPoints[a])*((float)yPoints[b] - (float)yPoints[a])) /
(((float)xPoints[b] - (float)xPoints[a])*((float)xPoints[b] - (float)xPoints[a]) +
((float)yPoints[b] - (float)yPoints[a])*((float)yPoints[b] - (float)yPoints[a]));

        x = (float)xPoints[a] + u * ((float)xPoints[b] - (float)xPoints[a]);

        y = (float)yPoints[a] + u * ((float)yPoints[b] - (float)yPoints[a]);

        return ((x - (float)xPoints[p])*(x - (float)xPoints[p]) + (y - (float)yPoints[p])*(y -
(float)yPoints[p]));

    }


    public int farthestpoint(int a, int b, ArrayList<Integer>al)

    {

        float maxD, dis;

        int maxP, p;

        maxD = -1;

        maxP = -1;

        for ( int i = 0; i < al.size(); i++ )

        {

            p = al.get(i);

            if ( (p == a) || (p == b) )
```

```
                continue;
            dis = distance(a, b, p);
            if ( dis > maxD )
            {
                maxD = dis;
                maxP = p;
            }
        }
        return maxP;
    }

    public void quickhull(int a, int b, ArrayList<Integer>al)
    {
        //System.out.println("a:"+a+",b:"+b+" size: "+al.size());
        if ( al.size() == 0 )
            return;

        int c, p;

        c = farthestpoint(a, b, al);

        ArrayList<Integer> al1 = new ArrayList<Integer>();
        ArrayList<Integer> al2 = new ArrayList<Integer>();

        for ( int i=0; i<al.size(); i++ )
        {
            p = al.get(i);
            if ( (p == a) || (p == b) )
                continue;
            if ( right(a,c,p) > 0 )
                al1.add(p);
            else if ( right(c,b,p) > 0 )
                al2.add(p);
        }

        quickhull(a, c, al1);
        xPoints2[num] = xPoints[c];
        yPoints2[num] = yPoints[c];
        num++;
        quickhull(c, b, al2);
    }

    //Generates the convex hull
    public void quickconvexhull()
    {
        // random
        int x, y;
        /*
        for ( int i = 0; i < pNum; i++ )
        {
            xPoints[i] = p1 + rnd.nextInt(w-100);
            yPoints[i] = p1+ rnd.nextInt(h-100);
        }
            */
        // find two points: right (bottom) and left (top)
        int r, l;
        r = l = 0;
        for ( int i = 1; i < pNum; i++ )
        {
            if ( ( xPoints[r] > xPoints[i] ) || ( xPoints[r] == xPoints[i] && yPoints[r] >
    yPoints[i] ))
                r = i;
            if ( ( xPoints[l] < xPoints[i] ) || ( xPoints[l] == xPoints[i] && yPoints[l] <
    yPoints[i] ))
                l = i;
        }

        ArrayList<Integer> al1 = new ArrayList<Integer>();
        ArrayList<Integer> al2 = new ArrayList<Integer>();

        int upper;
        for ( int i = 0; i < pNum; i++ )
        {
            if ( (i == l) || (i == r) )
                continue;
            upper = right(r,l,i);
            if ( upper > 0 )
                al1.add(i);
                else if ( upper < 0 )
                        al2.add(i);
        }

        xPoints2[num] = xPoints[r];
        yPoints2[num] = yPoints[r];
        num++;
        quickhull(r, l, al1);
        xPoints2[num] = xPoints[l];
        yPoints2[num] = yPoints[l];
        num++;
        quickhull(l, r, al2);
}

public void paint(Graphics g)
{
//Dijistras Algorithm parameters

        //  Set background as grey
        g.setColor(Color.lightGray);
        // Set the dimentions of the plane
        //g.fillRect(0,0,w-1,h-1);
        g.fillRect(0,0,600,600);

        // Way points of the phase
        g.setColor(Color.yellow);
        for ( int i = 0; i < dNum; i++ )
        {
                g.fillOval(xdijPoints[i]-2,ydijPoints[i]-2, 4,4);
        }

        // Path along the dijistras shortest route - joined by blue lines
        g.setColor(Color.blue);
        for ( int i = 0; i < size_of_path_list -1; i++ )
        {
                g.drawLine(xdijpathPoints[i], ydijpathPoints[i], xdijpathPoints[i+1],
ydijpathPoints[i+1] );
        }

        // THe points along the shortest path are highlighted in green - indicating GO
        g.setColor(Color.green);
        for ( int i = 0; i < size_of_path_list; i++ )
        {
                g.drawOval(xdijpathPoints[i]-5,ydijpathPoints[i]-5, 10,10);
        }

  //Convex hull algorithm parameters
        // IEDs
        g.setColor(Color.blue);
        for ( int i = 0; i < pNum; i++ )
        {
                g.fillOval(xPoints[i]-2,yPoints[i]-2, 4,4);
}

// Convex hull
g.setColor(Color.black);
g.drawPolygon(xPoints2, yPoints2, num);

// Red circles around the points that form the convex hull - indicating danger
g.setColor(Color.red);
for ( int i = 0; i < num; i++ )
{
        g.drawOval(xPoints2[i]-5,yPoints2[i]-5, 10,10);
}

//hull way points
g.setColor(Color.green);
for ( int i = 0; i < hull_path_count ; i++ )
{
        g.drawOval(xfinalhullpath[i]-5,yfinalhullpath[i]-5, 10,10);
}

//lines for the complete convex hull
g.setColor(Color.black);
g.drawPolygon(xPoints2, yPoints2, num);

//line for the path through the convex hull
g.setColor(Color.blue);
```

```
            for ( int i = 0; i < hull_path_count -1; i++ )
            {
                        g.drawLine(xfinalhullpath[i], yfinalhullpath[i], xfinalhullpath[i+1],
yfinalhullpath[i+1] );
            }


    //Final map generation (Drawing 3 lines to complete the map)
            g.setColor(Color.lightGray);
            g.drawLine(xinit, yinit, xend, yend);


            g.setColor(Color.blue);
            g.drawLine(xinit, yinit, xhullinit, yhullinit);
            g.drawLine(xhullend, yhullend, xend, yend);
    }
}
```